

1. What is the order of execution for Main Method, Static block, Constructor, Instance Block?

- Static Block
- Main Method
- Constructor
- Instance Method

2. Can we overload Main Method?

Yes we can overload the Main Method

3. Can we override main method?

No, we cannot override main method of java because a static method cannot be overridden. The static method in java is associated with class.

When superclass and subclass contain the same method including parameters and if they are static. The method in the superclass will be hidden by the one that is in the subclass.

This mechanism is known as **method hiding** in short, though super and subclasses have methods with the same signature if they are static, it is not considered as overriding.

4. Can we create the Object for final class?

Yes we can create the object for Final Class

5. Can we create private classes?

We cannot create the private class. Will get this error Illegal modifier for the class PrivateClass; only public, abstract & final are permitted

6. Can we create the protected classes?

We cannot create the protected class. Will get this error Illegal modifier for the class Protected Class; only public, abstract & final are permitted

7. Explain different types of access modifiers in Java?

| Modifier | Class | Package | Subclass | Global |
|-----------|-------|---------|----------|--------|
| Public | Yes | Yes | Yes | Yes |
| Protected | Yes | Yes | Yes | No |
| Default | Yes | Yes | No | No |
| Private | Yes | No | No | No |

8. Explain about non access modifiers in Java?

Non Access Modifiers

- final
- static
- abstract
- strictfp
- native
- synchronized
- transient
- volatile

www.JAVA95.com

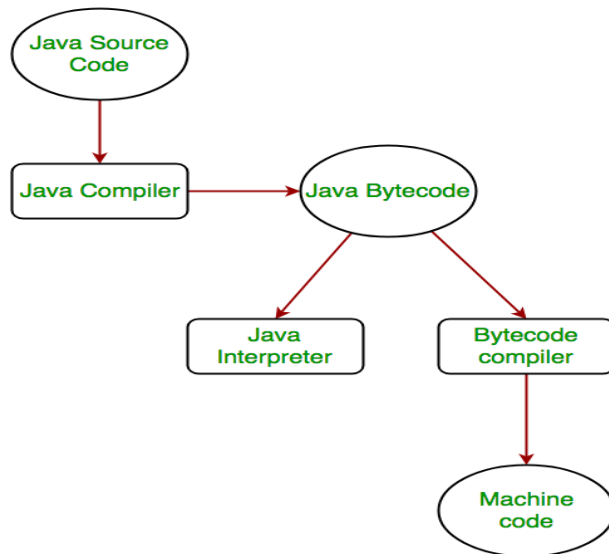
9. Explain Java reserved words?

| | | | | |
|----------|---------|------------|--------------|-----------|
| abstract | do | if | private | this |
| assert | double | implements | protected | throw |
| boolean | else | import | public | throws |
| break | enum | instanceof | return | transient |
| byte | extends | int | short | true |
| case | false | interface | static | try |
| catch | final | long | strictfp | void |
| char | finally | native | super | volatile |
| class | float | new | switch | while |
| const | for | null | synchronized | continue |
| default | goto | package | | |

10. Why Java is platform independent?

The meaning of platform-independent is that the java compiled code (byte code) can run on all operating systems.

When the Java program runs in a particular machine it is sent to java compiler, which converts this code into intermediate code called byte code. This byte code is sent to Java virtual machine (JVM) which resides in the RAM of any operating system. JVM recognizes the platform it is on and converts the byte codes into native machine code. Hence java is called platform independent language.



11. Is JVM is platform independent/dependent?

JAVA is platform-independent language, the JVM is platform-dependent.

12. Is Java a compiler or interpreter language?

Java can be considered both a compiled and an interpreted language because its source code is first compiled into a binary byte-code. This byte-code runs on the Java Virtual Machine (JVM), which is usually a software-based interpreter.

13. What is Constructor and it's Types?

Constructor is a special data member that will be executed during the object creation. We have two types of constructors.

Default Constructor: A constructor is called "Default Constructor" when it doesn't have any parameter.

Parameterized Constructor: A constructor which has a specific number of parameters is called a parameterized constructor.

14. What will happens if we did not provide any constructor?

If there is no constructor in a class, compiler automatically creates a default constructor.

15. What is the purpose of a default constructor?

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

16. Why will use the parameterized constructor?

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

17. Difference between Constructor and Methods?

| Java Constructor | Java Method |
|---|--|
| Constructor is used to initialize the state of an object. | Method is used to expose the behavior of an object. |
| Constructor doesn't have any return type. | Method must have a return type. |
| Constructor is invoked implicitly. | Method is invoked explicitly. |
| The name of the Constructor must be the same as the class name. | The name of the method may or may not be same as the class name. |

18. Can we overload private Methods?

Yes, we can overload private methods in Java but, you can access these from the same class.

19. Can we override private methods?

No, we cannot override private or static methods in Java. Private methods in Java are not visible to any other class which limits their scope to the class in which they are declared.

20. Can we overload final Methods?

Yes, we can overload final methods in Java

21. Can we override final methods?

No, we cannot override final methods in Java. A final method declared in the Parent class cannot be overridden by a child class. If we try to override the final method, the compiler will throw an exception at compile time. Because if we are declaring any method with the final keyword, we are indicating the JVM to provide some special attention and make sure no one can override it

22. Can we overload protected Methods?

Yes, we can overload protected methods in Java

23. Can we override the protected methods?

Yes, the protected method of a superclass can be overridden by a subclass. If the superclass method is protected, the subclass overridden method can have protected or public (but not default or private) which means the subclass overridden method cannot have a weaker access specifier.

24. Can we overload the default methods?

Yes, we can overload default methods in Java

25. Can we override the default methods?

Yes, we can override default methods in Java

26. Can we overload the Constructor?

Yes we can overload the constructor

27. Can we override the Constructor?

Constructor looks like method but it is not. It does not have a return type and its name is same as the class name.

But, a constructor cannot be overridden. If you try to write a super class's constructor in the sub class compiler treats it as a method and expects a return type and generates a compile time error.

28. Difference between Overloading and Overriding?

| S.NO | METHOD OVERLOADING | METHOD OVERRIDING |
|------|---|---|
| 1. | Method overloading is a compile time polymorphism. | Method overriding is a run time polymorphism. |
| 2. | It help to rise the readability of the program. | While it is used to grant the specific implementation of the method which is already provided by its parent class or super class. |
| 3. | It is occur within the class. | While it is performed in two classes with inheritance relationship. |
| 4. | Method overloading may or may not require inheritance. | While method overriding always needs inheritance. |
| 5. | In this, methods must have same name and different signature. | While in this, methods must have same name and same signature. |
| 6. | In method overloading, return type can or can not be be same, but we must have to change the parameter. | While in this, return type must be same or co-variant. |

29. Explain the proper objects to be created for below Inheritance example?

```
MethodOverriding.java
1 package com.interview.preparation;
2 class Parent{
3
4     void display() {
5         System.out.println("In Parent class display method");
6     }
7 }
8 class Child extends Parent{
9     void display() {
10        System.out.println("In child class display method");
11    }
12 }
13 public class MethodOverriding {
14
15     public static void main(String[] args) {
16         Parent parent = new Parent();
17         parent.display(); //In Parent class display method
18
19         Child child = new Child();
20         child.display(); //In child class display method
21
22         Parent p1 = new Child();
23         p1.display(); //In child class display method
24
25         //Child c1 = new Parent(); // Compile error Type mismatch: cannot convert from Parent to Child
26
27         Child c2 = (Child) new Parent ();
28         c2.display(); //During execution will get ClassCastException
29     }
30 }
```

30. Explain the exception handling possibilities Inheritance?

- If SuperClass does not declare an exception, then the SubClass can only declare unchecked exceptions, but not the checked exceptions.
- If SuperClass declares an exception, then the SubClass can only declare the child exceptions of the exception declared by the SuperClass, but not any other exception.
- If SuperClass declares an exception, then the SubClass can declare without exception.

31. Is Java “pass-by-reference” or “pass-by-value”

Java is always “pass-by-value”. However, when we pass the value of an object, we pass the reference to it because the variables store the object reference, not the object itself. But this isn’t “pass-by-reference.”

32. What will happen when we raise the exception but not catching it?

When an exception occurred, if you don’t handle it, the program terminates abruptly and the code past the line that caused the exception will not get executed.

33. What will happens if you raise the NullPointerException but catching the ArrayIndexOutOfBoundsException?

```
ExceptionHandling.java
1 package com.interview.preparation;
2
3 import com.interview.preparation.beans.Student;
4
5 public class ExceptionHandling {
6
7     public static void main(String[] args) {
8         Student student = null;
9         try {
10            System.out.println(student.getName());
11        } catch (ArrayIndexOutOfBoundsException e) {
12            System.out.println("Exception ");
13        }
14    }
15
16 }
17
```

Program will not go to the Catch block and it will terminate at **line number 10** and it will through **NullPointerException**.

34. How to create Java immutable classes?

Immutable class means that once an object is created, we cannot change its content. In Java, all the wrapper classes (like Integer, Boolean, Byte, Short) and String class is immutable. We can create our own immutable class as well.

Following are the requirements:

- ✓ The class must be declared as final (So that child classes can't be created)
- ✓ Data members in the class must be declared as private (So that direct access is not allowed)
- ✓ Data members in the class must be declared as final (So that we can't change the value of it after object creation)
- ✓ A parameterized constructor should initialize all the fields performing a deep copy (So that data members can't be modified with object reference)
- ✓ Deep Copy of objects should be performed in the getter methods (To return a copy rather than returning the actual object reference)
- ✓ No setters (To not have the option to change the value of the instance variable)

Example:

```

public final class ImmutableClass {

    private final String pancardNumber;

    public ImmutableClass(String pancardNumber) {
        this.pancardNumber = pancardNumber;
    }

    public String getPancardNumber() {
        return pancardNumber;
    }

    public static void main(String[] args) {
        ImmutableClass immutableClass = new ImmutableClass("1234567687");
        System.out.println(immutableClass.getPancardNumber()); //1234567687
    }
}

```

35. Difference b/w final, finally and finalize?

| Final | Finally | Finalize |
|---|---|---|
| Final is a "Keyword" and "access modifier" in Java | Finally is a "block" in Java | Finalize is a "method" in Java |
| Final is a keyword applicable to classes, variables and methods. | Finally is a block that is always associated with try and catch block. | Finalize () is a method applicable to objects. |
| (1) Final variable becomes constant, and it can't be reassigned. (2) A final method can't be overridden by the child class. (3) Final Class cannot be extended. | A "finally" block, clean up the resources used in "try" block. | Finalize method performs cleans up activities related to the object before its destruction. |
| Final method is executed upon its call. | "Finally" block executes just after the execution of "try-catch" block. | Finalize () method executes just before the destruction of the object. |

36. Explain about Garbage Collections in JAVA?

Garbage Collection in Java is a process by which the programs perform memory management automatically. The Garbage Collector (GC) finds the unused objects and deletes them to reclaim the memory. In Java, dynamic memory allocation of objects is achieved using the new operator that uses some memory and the memory remains allocated until there are references for the use of the object.

When there are no references to an object, it is assumed to be no longer needed, and the memory, occupied by the object can be reclaimed. There is no explicit need to destroy an object as Java handles the de-allocation automatically.

The technique that accomplishes this is known as Garbage Collection. Programs that do not de-allocate memory can eventually crash when there is no memory left in the system to allocate. These programs are said to have memory leaks.

Garbage collection in Java happens automatically during the lifetime of the program, eliminating the need to de-allocate memory and thereby avoiding memory leaks.

In C language, it is the programmer's responsibility to de-allocate memory allocated dynamically using free() function. This is where Java memory management leads.

Note: All objects are created in Heap Section of memory.

37. When does an Object becomes eligible for Garbage collection in Java?

When an object created in Java program is no longer reachable or used it is eligible for garbage collection.

38. Explain the ways to make an object eligible for garbage collection in Java?

By nullifying the reference variable

```
-----  
Student s1 = new Student( );  
Student s2 = new Student( );  
s1 = null;  
s2 = null;
```

By reassigning the reference variable

```
-----  
Student s1 = new Student( );  
Student s2 = new Student( );  
  
//Not eligible for garbage collections  
s1 = new Student( );  
  
s2=s1
```

By creating objects inside a method

```
-----  
class Test  
{  
    public static void main(String[ ] args)
```

```

{
    m1( );
}

public static void m1( )
{
    Student s1 = new Student( );
    Student s2 = new Student( );
}
}

```

39. What is the difference between ParNew and DefNew Young Generation Garbage collector?

ParNew and DefNew are two young generation garbage collector. ParNew is a multi-threaded GC used along with concurrent Mark Sweep while DefNew is single-threaded GC used along with Serial Garbage Collector.

40. Can we force Garbage collection in Java?

Yes we can force the garbage collection by using `System.gc()` and `Runtime.gc()` but there is no guarantee of object to be garbage collected.

41. In which case will force the Garbage Collector Manually?

One possible use case is if you load really large things, they'll end up in the Large Object Heap which will go straight to Gen 2, though again Gen 2 is for long lived objects because it collects less frequently. If you know that you are loading short lived objects into Gen 2 for any reason, you could clear them out more quickly to keep your Gen 2 smaller and it's collections faster.

42. Does Garbage collection occur in permanent generation space in JVM?

Yes. Metadata such as classes and methods are stored in the Permanent Generation. Classes that are no longer in use may be garbage collected from the Permanent Generation.

This is why the correct sizing of PermGen space is essential to avoid frequent full GC. You can control the size of PermGen space by JVM options `-XX:PermGenSize` and `-XX:MaxPermGenSize`.

43. How to you monitor garbage collection activities?

You can monitor garbage collection activities, either offline or in real-time. You can use tools like JConsole and VisualVM VM with its Visual GC plug-in to monitor real-time garbage collection activities, and memory status of JVM

44. Which part of the memory is involved in Garbage Collection? Stack or Heap?

Heap

45. How many times does the garbage collector calls the finalize() method for an object?

Only Once

46. What happens if an uncaught exception is thrown from during the execution of the finalize() method of an object?

The exception will be ignored and the garbage collection (finalization) of that object terminates

47. What are the algorithms used by Garbage Collections?

Mark-Sweep algorithm:

The Mark-Sweep algorithm is the most common garbage collection algorithm, which performs two operations. It first marks the objects to be garbage-collected in the memory space and then clears the marked objects up from the heap.

Reference Counting Algorithm:

The Reference Counting Algorithm allocates a field in the object header to store the reference count of the object. If this object is referenced by another object, its reference count increments by one. If the reference to this object is deleted, the reference count decrements by one. When the reference count of this object drops to zero, the object will be garbage-collected.

48. What is meant by OutOfMemoryError?

Usually, this error is thrown when there is insufficient space to allocate an object in the Java heap. In this case, the garbage collector cannot make space available to accommodate a new object, and the heap cannot be expanded further. Also, this error may be thrown when there is insufficient native memory to support the loading of a Java class. In a rare instance, a **java.lang.OutOfMemoryError** may be thrown when an excessive amount of time is being spent doing garbage collection and little memory is being freed.

49. Explain the solutions to avoid the OutOfMemoryError?

- ✓ Nullify the object references.
- ✓ Close the Database connections
- ✓ Increase the Heap area size (JVM options "-Xmx512M")
- ✓ Take care of loops.
- ✓ Take care of array size declaration

50. When we will get the StackOverflowError?

Stack is used in Java for method execution, for every method call, a block of memory is created in the stack. The data related to method like parameters, local variables or references to objects are stored in this block.

When the method finishes its execution, this block is removed from the stack along with data stored in it. If a method keep calling other method recursively without returning back then at one point Stack will be full and there would not be any space left for allocating new stack block at that time you will encounter `StackOverflowError`.

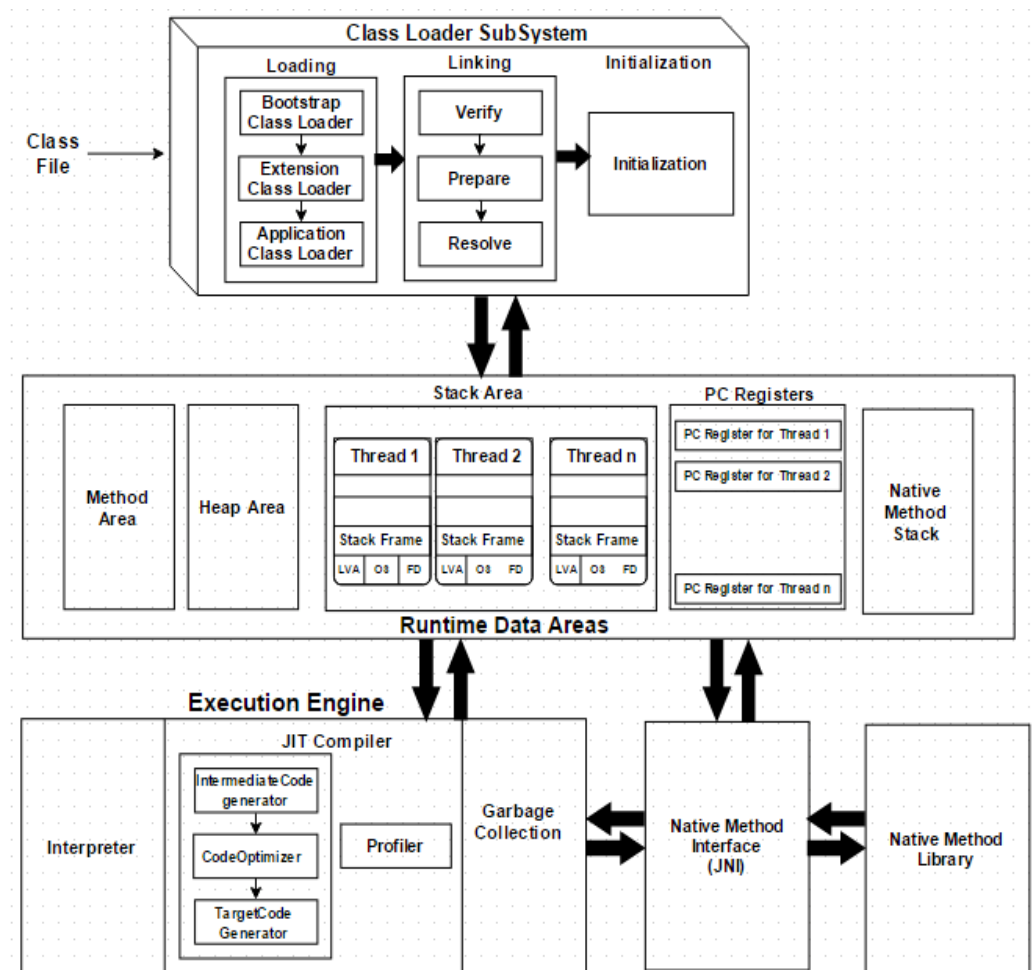
51. Explain about JDK?

JRE + Development Tools. Will use it to develop, compile and run the Java applications. Platform dependent.

52. Explain about JRE?

JVM + Class Libraries. Will use it to run the Java applications. Platform dependent

53. Explain JVM architecture?



Class loader:

Class loader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the class loader. There are three built-in class loaders in Java.

Bootstrap ClassLoader: It loads the rt.jar file which contains all class files of Java Standard Edition like java.lang package classes, java.net package classes, java.util package classes, java.io package classes, java.sql package classes etc.

Extension ClassLoader: Responsible loading remaining jars available in JRE folder (**jre/lib/ext**)

Application ClassLoader: It loads the classfiles from classpath. By default, classpath is set to current directory.

Loading:

Responsible to read the .class from hard disk and load it to Method area

Linking:

Performs verification, preparation, and (optionally) resolution. It has below three phases.

Verification:

It ensures the correctness of the .class file i.e. it checks whether this file is properly formatted and generated by a valid compiler or not. If verification fails, we get run-time exception `java.lang.VerifyError`. This activity is done by the component `ByteCodeVerifier`. Once this activity is completed then the class file is ready for compilation.

Preparation:

JVM allocates memory for class variables and initializing the memory to default values.

Resolution:

It is the process of replacing symbolic references from the type with direct references. It is done by searching into the method area to locate the referenced entity.

Initialization:

In this phase, all static variables are assigned with their values defined in the code and static block(if any). This is executed from top to bottom in a class and from parent to child in the class hierarchy.

Memory Areas:

Method Area: Methods Info, Variables Info, Constructor Info, Modifiers Info, Constant Pool Info

Heap Area: It is the runtime data area in which objects are allocated.

Stack Area: For every thread, JVM creates one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which stores methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminates, its run-time stack will be destroyed by JVM. It is not a shared resource.

Pc Register Area: Store address of current execution instruction of a thread. Obviously, each thread has separate PC Registers

Native Method Stack Area: It stores native method information.

Execution Engine:

Execution engine executes the “.class” (bytecode). It reads the byte-code line by line, uses data and information present in various memory area and executes instructions. It can be classified into three parts:

1. Interpreter

It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.

2. JIT Compiler

It is used to increase the efficiency of an interpreter. It compiles the entire bytecode and changes it to native code so whenever the interpreter sees repeated method calls, JIT provides direct native code for that part so re-interpretation is not required, thus efficiency is improved.

3. Garbage Collector

It destroys un-referenced objects. For more on Garbage Collector, refer

Native Method Libraries:

It is a collection of the Native Libraries(C, C++) which are required by the Execution Engine.

Java Native Interface (JNI):

It is an interface that interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution. It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.

54. Explain about Heap Memory Generations?

Young Generation:

Newly created objects start in the Young Generation. If young space is full then garbage collector will be called and memory will be released.

Old Generation:

This space holding those objects which survived after garbage collection from Young Generation. When objects are garbage collected from the Old Generation, it is a major garbage collection event.

Permanent Generation:

Metadata such as classes and methods are stored in the Permanent Generation. Classes that are no longer in use may be garbage collected from the Permanent Generation.

55. Explain Java OOPS concepts?

Class: It's a collections of objects/ Blue print of object. It's a kind of drawing an object.

Object: It's a real world entity. It has State and behaviour.

Ex: Class Dog

State: Size, Breed, age, color

Behaviour: eat(), sleep (), run (), bark()

Abstraction: Its a process of hiding the internal details and showing the functionality.

Example Car.

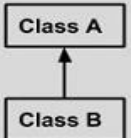
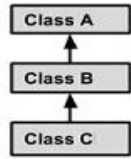
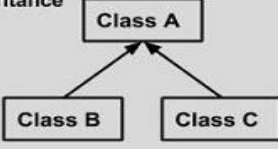
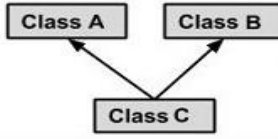
Encapsulation: Wrapping up the data into an single unit. As in encapsulation, the data in a class is hidden from other classes, so it is also known as data-hiding.

Example Capsule.

Inheritance: Acquiring all the properties and behaviors of a parent object

Polymorphism: Ability of having more than one form.

56. Explain different types of Inheritances in JAVA?

| | |
|---|---|
| Single Inheritance  <pre>graph BT; B[Class B] --> A[Class A]</pre> | <pre>public class A { } public class B extends A { }</pre> |
| Multi Level Inheritance  <pre>graph BT; C[Class C] --> B[Class B]; B --> A[Class A]</pre> | <pre>public class A {} public class B extends A {.....} public class C extends B {.....}</pre> |
| Hierarchical Inheritance  <pre>graph BT; B[Class B] --> A[Class A]; C[Class C] --> A</pre> | <pre>public class A {} public class B extends A {.....} public class C extends A {.....}</pre> |
| Multiple Inheritance  <pre>graph BT; A[Class A] --> C[Class C]; B[Class B] --> C</pre> | <pre>public class A {} public class B {.....} public class C extends A,B { } // Java does not support mutiple Inheritance</pre> |

Single Inheritance: In single inheritance, subclasses inherit the features of one superclass

Multilevel Inheritance: In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.

Hierarchical Inheritance: one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived class B, C and D.

Multiple Inheritance (Through Interfaces): In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes. Please note that Java does not support multiple inheritances with classes. In java, we can achieve multiple inheritances only through Interfaces.

57. Why Java does not support Multiple Inheritance?

The reason behind this is to prevent ambiguity.

Consider a case where class B extends class A and Class C and both class A and C have the same method display().

Now java compiler cannot decide, which display method it should inherit. To prevent such situation, multiple inheritances is not allowed in java.

58. Explain different types of Polymorphism?

There are two types of polymorphism in Java:

Compile-time polymorphism: It is a process in which a call to an method is resolved at compile time

Ex: Method Overloading

Runtime polymorphism/Dynamic Method Dispatch: It is a process in which a call to an overridden method is resolved at runtime

Ex: Method Overriding

59. What is the need of Wrapper Classes and its uses?

Wrapper classes are used to wrap the primitive data types to Java Objects.

Autoboxing: Automatic conversion of primitive types to the object type

Ex:

```
int a=20;
```

```
Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
```

Unboxing: It is just the reverse process of Autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type

Ex:

```
Integer a=new Integer(3);
```

```
int j=a;//unboxing, now compiler will write a.intValue() internally
```

Uses:

Change the value in Method: Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.

Serialization: We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.

Synchronization: Java synchronization works with objects in Multithreading.

java.util package: The java.util package provides the utility classes to deal with objects.

Collection Framework: Java collection framework works with objects only.

60. Differences between this and super?

| this() | super() |
|---|---|
| 1. this() represents the current instance of a class | 1. super() represents the current instance of a parent/base class |
| 2. Used to call the default constructor of the same class | 2. Used to call the default constructor of the parent/base class |
| 3. Used to access methods of the current class | 3. Used to access methods of the base class |
| 4. Used for pointing the current class instance | 4. Used for pointing the superclass instance |
| 5. Must be the first line of a block | 5. Must be the first line of a block |

61. Difference b/w static and instance methods?

| Instance Method | Static Method |
|--|--|
| Instance method are methods which require an object of its class to be created before it can be called. | Static methods are the methods in Java that can be called without creating an object of class. |
| Instance method can access the instance methods and instance variables directly. | Static methods can access the static variables and static methods directly. |
| Instance method can access static variables and static methods directly. | Static methods can't access instance methods and instance variables directly. They must use reference to object. |
| Instance method is not with static keyword | Static method is declared with static keyword |
| But instance methods exist as multiple copies depending on the number of instances created for that class. | Static method means which will exist as a single copy for a class. |

62. Explain Different ways to create a Java Object?

Using new Keyword:

```
-----
NewKeywordExample obj = new NewKeywordExample();
```

Using newInstance:

```
-----
Class cls = Class.forName("NewInstanceExample");
NewInstanceExample obj =
    (NewInstanceExample) cls.newInstance();
```

Using newInstance() method of Constructor class (Reflection) :

```
-----
Constructor<ReflectionExample> constructor =
    ReflectionExample.class.getDeclaredConstructor();
ReflectionExample r = constructor.newInstance();
```

Using clone() method:

```
-----
CloneExample obj1 = new CloneExample();
CloneExample obj2 = (CloneExample) obj1.clone();
```

```
@Override
protected Object clone() throws CloneNotSupportedException
{
    return super.clone();
}
```

Using deserialization:

```
DeserializationExample d =new DeserializationExample("GeeksForGeeks");  
FileOutputStream f = new FileOutputStream("file.txt");  
ObjectOutputStream oos = new ObjectOutputStream(f);  
oos.writeObject(d);
```

63. Difference b/w Abstract class and Interface?

| Abstract Classes | Interfaces |
|---|--|
| The abstract keyword is used to declare abstract class. | The interface keyword is used to declare interface. |
| Abstract class can have abstract and non-abstract methods. | Interface can have only abstract methods. Since Java 8, it can have default and static methods also. |
| Abstract class doesn't support multiple inheritance. | Interface supports multiple inheritance. |
| Abstract class can have final, non-final, static and non-static variables. | Interface has only static and final variables. |
| Abstract class can provide the implementation of interface. | Interface can't provide the implementation of abstract class. |
| An abstract class can extend another Java class and implement multiple Java interfaces. | An interface can extend another Java interface only. |
| A Java abstract class can have class members like private, protected, etc. | Members of a Java interface are public by default. |

64. When to use Abstract Class and Interface?

If you don't know the clear requirement then will go for the Abstract classes

If we are clear about the requirement and it will not change in future will go for interfaces

65. Why we need the Interfaces?

It is used to achieve abstraction.

By interface, we can support the functionality of multiple inheritance.

It can be used to achieve loose coupling.

66. Explain about Marker Interface?

Interface with no methods is called as Marker Interface.

Java has many built-in marker interfaces, such as **Serializable**, **Cloneable**

67. What is the need of Marker Interface?

It is used to indicate or inform the JVM that a class implementing this interface will have some special behaviour. An efficient way to classify code can be achieved using the marker interface. Examples of such an interface are: Serializable, Cloneable and Remote Interface.

68. How multiple inheritance possible with Interfaces?

An interface contains variables and methods like a class but the methods in an interface are abstract by default unlike a class. Multiple inheritance by interface occurs if a class implements multiple interfaces or also if an interface itself extends multiple interfaces.

```
interface InterfaceOne
{
    public void disp();
}
interface InterfaceTwo
{
    public void disp();
}
public class Main implements InterfaceOne,InterfaceTwo
{
    @Override
    public void disp()
    {
        System.out.println("display() method implementation");
    }
    public static void main(String args[])
    {
        Main m = new Main();
        m.disp();
    }
}
```

O/P: display() method implementation

69. Can we create the Object for Abstract class? How to Access the Non Abstract methods of Abstract Class?

No, we can't create an object of an abstract class. But we can create a reference variable of an abstract class. The reference variable is used to refer to the objects of derived classes (subclasses of abstract class).

We can access the abstract class non-abstract methods in below way,

```

abstract class Test{
    abstract void show();

    public void display() {
        System.out.println("In Parent class display method");
    }
}

public class AbstractClassInstantion extends Test {

    @Override
    void show() {
        System.out.println("Showing in implementation class");
    }

    public static void main(String[] args) {
        Test test = new AbstractClassInstantion();

        //Way to invoke the abstract class no abstract methods
        test.display(); //In Parent class display method
        test.show(); //Showing in implementation class
    }
}

```

70. Can we create the Object for Interface?

No, you cannot instantiate an interface. Generally, it contains abstract methods (except default and static methods introduced in Java8), which are incomplete.

Still if you try to instantiate an interface, a compile time error will be generated saying "MyInterface is abstract; cannot be instantiated".

71. Why String is Immutable?

Immutable simply means unmodifiable or unchangeable

Once string object is created its data or state can't be changed but a new string object is created.

When we create a string in java like `String s1="hello";` then an object will be created in string pool(hello) and s1 will be pointing to hello. Now if again we do `String s2="hello";` then another object will not be created but s2 will point to hello because JVM will first check if the same object is present in string pool or not. If not present then only a new one is created else not.

Because java uses the concept of string literal. Suppose there are 5 reference variables, all refers to one object "sachin". If one reference variable changes the value of the object, it will be affected to all the reference variables. That is why string objects are immutable in java.

Few tricks of Strings Java Program.

```
StringImmutable.java
1 package com.interview.preparation;
2
3 public class StringImmutable {
4
5     public static void main(String[] args) {
6         String s="Sachin";
7         String s1 = "Sachin";
8
9         System.out.println(s + " " + s1); //Sachin Sachin
10
11         s = "tendulkar";
12         System.out.println(s + " " + s1); //tendulkar Sachin
13
14         s.concat(" Tendulkar");
15         s1.concat(" Tendulkar");
16         System.out.println(s); //tendulkar Sachin
17
18         s = s.concat(" Tendulkar");
19         System.out.println(s); //tendulkar Tendulkar
20     }
21 }

```

72. Explain the different ways to create the String Object?

Assigning a string value wrapped in " " to a String type variable.

```
String message = "Hello";
```

Creating an object of the String class using the new keyword by passing the string value as a parameter of its constructor.

```
String message = new String ("Hello");
```

73. Difference b/w creating the Object from String literal and new Object?

Whenever we create a string literal, the Java Virtual Machine (JVM) first checks in the "string constant pool",

There are two cases:

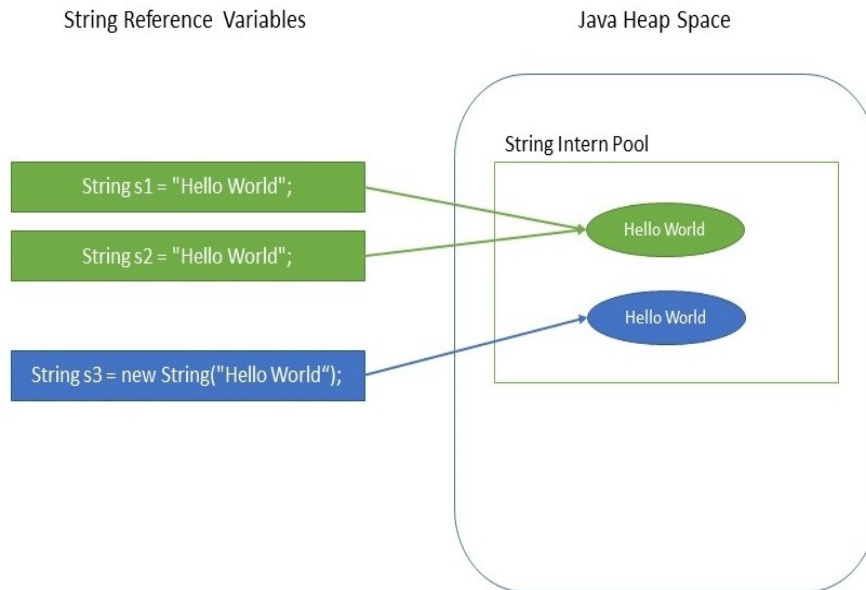
- If, String Literal already exists in the pool then, in that case, a reference to the pooled instance is returned (i.e. no new instance is created for the existing String Literal).
- String Literal is not already existed in the pool then, in that case, a new string instance is created and placed in the pool (i.e. new instance is created for the new String Literal).
- String objects are stored in a special memory area known as "String Constant Pool".

Whenever we create a String object by using "new" keyword,

String str = new String("Java Programming");

The Java Virtual Machine (JVM) creates a new string object in "Heap Memory" and the literal "Java Programming" will be placed in "String Constant Pool" and the variable "str" will refer to the string object placed in "Heap Memory".

When we create string objects by using "new" keyword so the objects are stored in a special memory area known as "Heap".



Few Tricky Tips:

```
StringsCompansion.java  ⌕
1 package com.interview.preparation;
2
3 public class StringsCompansion {
4
5     public static void main(String[] args) {
6         String s="Sachin";
7         String s1 = "Sachin";
8
9         System.out.println(s==s1); //true
10
11         String s3= new String("Sachin");
12         String s4 = new String("Sachin");
13
14         System.out.println(s3.hashCode() + " " + s4.hashCode()); //-1826113608 -1826113608
15         System.out.println(s3==s4); //false
16         System.out.println(s3.equalsIgnoreCase(s4)); //true
17
18         String s5= new String("Sachin").intern();
19         String s6 = new String("Sachin").intern();
20
21         System.out.println(s5.hashCode() + " " + s6.hashCode()); //-1826113608 -1826113608
22
23         System.out.println(s5==s6); //true
24         System.out.println(s5.equalsIgnoreCase(s6)); //true
25     }
26 }
```


74. How to put the String objects into String constant pool?

Java by default doesn't put all String objects into the String pool, instead, it gives you the flexibility to explicitly store any arbitrary object in the String pool. You can put any object to the String pool by calling the `intern()` method of `java.lang.String` class. Though, when you create using String literal notation of Java, it automatically calls `intern()` method to put that object into String pool, provided it was not present in the pool already.

75. Difference b/w String, String Builder and StringBuffer?

| Factor / Class | String | StringBuffer | StringBuilder |
|----------------|-----------------|--------------|-----------------|
| Mutability | Immutable | Mutable | Mutable |
| Thread Safety | Not thread safe | Thread safe | Not thread safe |
| Performance | Very high | Moderate | Very high |

76. What is the difference between `equals()` method and `==` operator?

The `equals()` method matches content of the strings whereas `==` operator matches object or reference of the strings.

77. Is String class final?

Yes

78. Can we use String in switch case?

Java 7 extended the capability of switch case to use Strings also, earlier Java versions don't support this.

79. Why Char array is preferred over String for storing password?

String is immutable in Java and stored in String pool. Once it's created it stays in the pool until unless garbage collected, so even though we are done with password it's available in memory for longer duration and there is no way to avoid it. It's a security risk because anyone having access to memory dump can find the password as clear text.

If we use a char array to store password, we can set it to blank once we are done with it. So we can control for how long it's available in memory that avoids the security threat with String.

80. Do all properties of an Immutable Object need to be final?

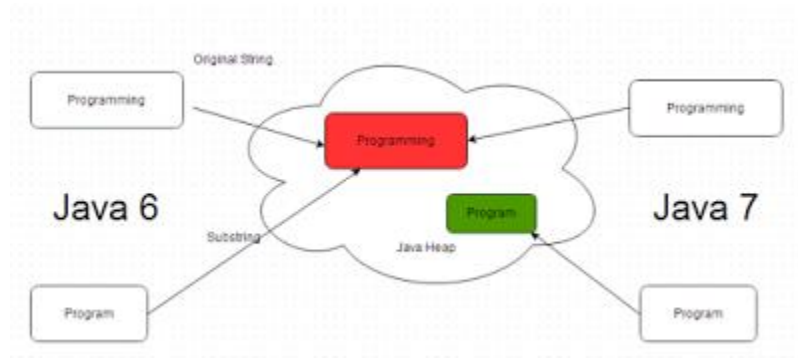
Not necessarily, as stated above you can achieve same functionality by making the member non-final but private and not modifying them except in a constructor. Don't provide setter methods for them and if it is a mutable object, then don't ever leak any reference for that member.

Remember making a reference variable final, only ensures that it will not be reassigned to a different value, but you can still change individual properties of object, pointed by that reference variable.

81. How does the substring() method inside String works?

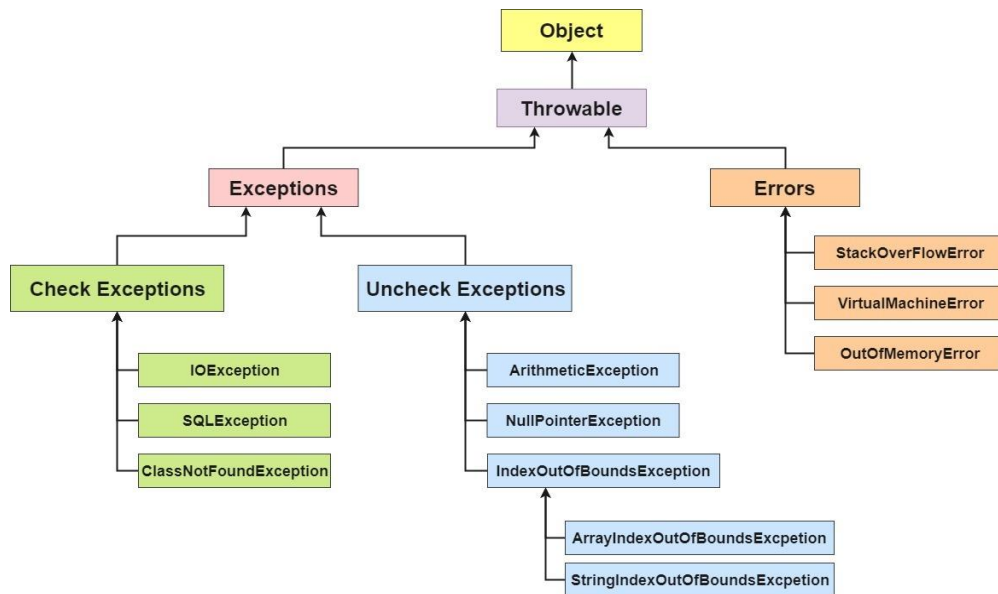
substring() method returns new string that is substring() of given string.

Until Java 1.7, substring holds the reference of the original character array, which means even a sub-string of 5 character long, can prevent 1GB character array from garbage collection, by holding a strong reference.



This issue is fixed in Java 1.7, where the original character array is not referenced anymore, but that change also made the creation of substring a bit more costly in terms of time. Earlier it was on the range of $O(1)$, which could be $O(n)$ in worst case on Java 7.

82. Explain the Exceptions Hierarchy?



Throwable: The java.lang.Throwable class is the root class of Java Exception hierarchy.

Exception: Exception is an abnormal condition.

Error: Error is irrecoverable

OutOfMemoryError
StackOverflowError
VirtualMachineError

Exception Types:

Checked Exceptions: Checked exceptions are checked at compile-time.

IOException
ClassNotFoundException
SQLException

Un Checked Exceptions: Un Checked exceptions are checked at run-time.

NullPointerException
ArithmeticException
NumberFormatException
IndexOutOfBoundsException
ArrayIndexOutOfBoundsException
StringIndexOutOfBoundsException

83. Explain the internal behaviour of Checked Exception, Unchecked exceptions and Error?

Checked Exceptions:

Checked exceptions are checked at compile-time. It means if a method is throwing a checked exception then it should handle the exception using try-catch block or it should declare the exception using throws keyword, otherwise the program will give a compilation error.

Example:

FileNotFoundException is a checked exception in Java. Anytime, we want to read a file from filesystem, Java forces us to handle error situation where file may not be present in place.

```
public static void main(String[] args)
{
    FileReader file = new FileReader("somefile.txt");
}
```

In above case, you will get compile time error with message – Unhandled exception type FileNotFoundException.

Unchecked Exceptions:

Unchecked exceptions are not checked at compile time. It means if your program is throwing an unchecked exception and even if you didn't handle/declare that exception, the program won't give a compilation error. Most of the times these exception occurs due to the bad data provided by user during the user-program interaction. It is up to the programmer to judge the conditions in advance, that can cause such exceptions and handle them appropriately. All Unchecked exceptions are direct sub classes of RuntimeException class.

Example:

```
class Example {
    public static void main(String args[])
    {
        int num1=10;
        int num2=0;
        /*Since I'm dividing an integer with 0
        * it should throw ArithmeticException
        */
        int res=num1/num2;
        System.out.println(res);
    }
}
```

If you compile this code, it would compile successfully however when you will run it, it would throw `ArithmeticException`. That clearly shows that unchecked exceptions are not checked at compile-time, they occurs at runtime.

Error

- ✓ Recovering from Error is not possible. The only solution to errors is to terminate the execution.
- ✓ You will not be able to handle the Errors using try-catch blocks. Even if you handle them using try-catch blocks, your application will not recover if they happen.
- ✓ Errors happen at run time. They will not be known to compiler.
- ✓ Errors are mostly caused by the environment in which application is running.
- ✓ All errors in java are unchecked type

84. What is meant by Concurrent Modification Exception? How to handle it?

If we try to perform the multiple operation on the same object will the Concurrent modification Exception. These kind of exception will occurs in Collections framework.

For Example:

```
public class ConcurrentModification {  
    public static void main(String[] args) {  
        List<Integer> list = new ArrayList<>();  
        list.add(1);  
        list.add(2);  
        list.add(3);  
        list.add(4);  
        list.add(5);  
  
        for (Integer integer : list) {  
            if(integer ==3) {  
                list.remove(integer); //Exception in thread "main" java.util.ConcurrentModificationException  
            }  
        }  
    }  
}
```

In the Above example we are performing the two operation loop iteration and object removal from object. While object removal will get the **Concurrent Modification Exception**.

To avoid such exception will go for the **CopyOnWriteArrayList**. Which is thread safe. `CopyOnWriteArrayList` class is introduced in JDK 1.5, which implements the `List` interface. It is an enhanced version of `ArrayList` in which all modifications (add, set, remove, etc) are implemented by making a fresh copy. It is found in `java.util.concurrent` package.

85. Difference between `ClassNotFoundException` and `NoClassDefFoundError` in Java?

ClassNotFoundException and NoClassDefFoundError both occur when class is not found at runtime.

ClassNotFoundException:

ClassNotFoundException occurs when you try to load a class at runtime using Class.forName() method requested classes are not found in classpath. Most of the time this exception will occur when you try to run application without updating classpath with JAR files. This exception is a checked Exception.

This exception also occurs when you have two class loaders and if a ClassLoader tries to access a class which is loaded by another classloader in Java. Java ClassLoader is a part of Java Runtime Environment that dynamically loads Java classes in JVM(Java Virtual Machine)

```
public class Example {  
  
    public static void main(String args[]) {  
        try  
        {  
            Class.forName("GeeksForGeeks");  
        }  
        catch (ClassNotFoundException ex)  
        {  
            ex.printStackTrace();  
        }  
    }  
}
```

NoClassDefFoundError:

NoClassDefFoundError occurs when class was present during compile time and program was compiled and linked successfully but class was not present during runtime. This error occurs when a class has some dependencies on another class and latter class changes after compilation of former class.

```
class A  
{  
    // some code  
}  
  
public class B  
{  
    public static void main(String[] args)
```

```
{  
    A a = new A();  
}  
}
```

When you compile the above program, two .class files will be generated. One is A.class and another one is B.class. If you remove the A.class file and run the B.class file, Java Runtime System will throw NoClassDefFoundError

86. What do the expression 1.0 / 0.0 will return? will it throw Exception? any compile-time error?

The simple answer to this question is that it will not throw ArithmeticException and return Double.INFINITY.

87. How to handle the Exceptions in Java?

The try-catch is the simplest method of handling exceptions. Put the code you want to run in the try block, and any exceptions that the code throws are caught by one or more catch blocks

88. What is the use of finally block in exception handling?

Any code that must be executed irrespective of occurrence of an exception is put in a finally block. In other words, a finally block is executed in all circumstances. For example, if you have an open connection to a database or file, use the finally block to close the connection even if the try block throws an exception.

89. How the uncaught exceptions handled in Java?

In java, assume that, if we do not handle the exceptions in a program. In this case, when an exception occurs in a particular function, then Java prints an exception message with the help of uncaught exception handler.

The uncaught exceptions are the exceptions that are not caught by the compiler but automatically caught and handled by the Java built-in exception handler.

Java programming language has a very strong exception handling mechanism. It allows us to handle the exception using the keywords like try, catch, finally, throw, and throws.

When an uncaught exception occurs, the JVM calls a special private method known as dispatchUncaughtException(), on the Thread class in which the exception occurs and terminates the thread. It is introduced in Java 5 Version.

In the MVC flow we can achieve this by using the GlobalExceptionHandler with @ControllerAdvice.

90. How to skip the finally block execution?

You cannot skip the execution of the final block. Still if you want to do it forcefully when an exception occurred, the only way is to call the `System.exit(0)` method, at the end of the catch block which is just before the finally block.

91. Difference Between `System.exit(0)`, `System.exit(-1)` and `System.exit(1)` in Java

- Status - `exit(0)` - indicates Successful termination
- Status - `exit(-1)` - indicates unsuccessful termination with Exception
- Status - `exit(1)` - indicates Unsuccessful termination

In an if-else decision, if our program is executed as we expected, and finally we need to stop the program, then we use `System.exit(0)`, and `System.exit(1)` is usually placed in the catch block. To catch the exception and need to stop the program, we use `System.exit(1)`. This status=1 is used to indicate that this program is abnormally exited.

92. How to create the User defined Exceptions?

Java provides us facility to create our own exceptions which are basically derived classes of `Exception`.

```
CustomException.java
1 package com.interview.preparation;
2
3 public class CustomException extends Exception {
4
5     /**
6      *
7      */
8     private static final long serialVersionUID = -706339475746296863L;
9
10    public CustomException(String message) {
11        super(message);
12    }
13
14    public CustomException(String message, Throwable throwable) {
15        super(message, throwable);
16    }
17
18
19 }
```

93. Differences between throw and throws?

| throw | throws |
|--|--|
| Java throw keyword is used to explicitly throw an exception. | Java throws keyword is used to declare an exception. |
| Checked exception cannot be propagated using throw only. | Checked exception can be propagated with throws. |
| Throw is followed by an instance. | Throws is followed by class. |
| Throw is used within the method. | Throws is used with the method signature. |
| You cannot throw multiple exceptions. | You can declare multiple exceptions e.g. public void method()throws IOException, SQLException. |

94. Can we have an empty catch block?

We can have an empty catch block but it's an example of bad programming.

95. Provide some Java Exception Handling Best Practices?

- ✓ Clean Up Resources in a Finally Block
- ✓ Always better to maintain exception specific catch blocks instead of single catch block with Exception.
- ✓ Throw Exceptions With meaningful Messages
- ✓ Don't ignore the exception by putting empty catch blocks
- ✓ Either log the exception or throw it but never do the both
- ✓ Never throw any exception from finally block
- ✓ Use finally blocks instead of catch blocks if you are not going to handle exception
- ✓ Always clean up after handling the exception
- ✓ Always include all information about an exception in single log message

96. What is a Thread?

It's a light weight process and separate path of execution

97. Advantages of threads?

- ✓ Parallel processing
- ✓ Less memory utilization
- ✓ Less CPU utilization
- ✓ Better communication

98. Explain the different ways to create the Thread?

By extending the Thread Class

```
ThreadClass.java
1 package com.interview.preparation;
2
3
4 public class ThreadClass extends Thread {
5
6     public void run() {
7         System.out.println("thread is running...");
8     }
9
10    public static void main(String[] args) {
11        ThreadClass threadClass = new ThreadClass();
12        threadClass.start();
13    }
14 }
15
16 }
```

By Implementing the Runnable Interface.

```
ThreadClass.java
1 package com.interview.preparation;
2
3
4 public class ThreadClass implements Runnable {
5
6     public void run() {
7         System.out.println("thread is running...");
8     }
9
10    public static void main(String[] args) {
11        ThreadClass threadClass = new ThreadClass();
12        Thread thread = new Thread(threadClass);
13        thread.start();
14    }
15 }
16
17 }
```

99. Which one is better extending Thread Class or Implementing Runnable Interface?

- ✓ When we extend Thread class, we can't extend any other class even we require and when we implement Runnable, we can save a space for our class to extend any other class in future or now.
- ✓ When we extend Thread class, each of our thread creates unique object and associate with it. When we implements Runnable, it shares the same object to multiple threads.

100. Difference b/w Yield, Join?

The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.

101. What is meant by daemon thread and how to make the thread to Daemon?

Daemon thread a low priority thread runs in background.

Ex: Garbage Collector

To make the thread to Daemon thread we need to write the below code.

```
thread.setDaemon(true);
```

102. Difference b/w Sleep and Wait?

Sleep(): This Method is used to pause the execution of current thread for a specified time in Milliseconds.

Wait(): This method is defined in object class. It tells the calling thread (a.k.a Current Thread) to wait until another thread invoke's the notify() or notifyAll() method for this object.

103. Difference b/w wait, notify and notifyAll?

wait(): This method is defined in object class. It tells the calling thread to wait until another thread invokes the notify() or notifyAll() method for this object.

notify() method sends the notification to only one thread among the multiple waiting threads which are waiting for lock.

notifyAll() methods in the same context sends the notification to all waiting threads instead of single one thread.

104. Why wait, notify and notifyAll are in Object class?

Locks are made available on per Object basis, which is another reason wait and notify is declared in Object class rather the Thread class.

To provide the Inter thread communication.

In general lock on acquired on the objects not on the threads. Threads have no specific knowledge of each other. They can run asynchronously and are independent. They just run, lock, wait and get notified. They do not need to know about the status of other threads. They just need to call notify method on an object

105. Can we start the thread twice?

No we cannot start the thread twice we will get the [ava.lang.IllegalThreadStateException](#)

106. Explain thread priorities?

In a Multi-threading environment, thread scheduler assigns processor to a thread based on priority of thread. Whenever we create a thread in Java, it always has some priority assigned to

it. Priority can either be given by JVM while creating the thread or it can be given by programmer explicitly.

MIN_PRIORITY - 1

MAX_PRIORITY - 10

NORM_PRIORITY - 5

```
ThreadDemo t1 = new ThreadDemo();
```

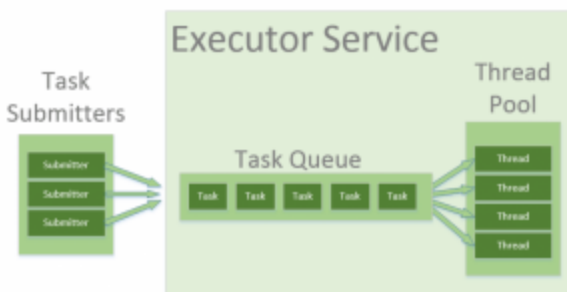
```
t1.setPriority(2);
```

107. Difference between preemptive scheduling and time slicing?

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

108. What is the need of thread pool?

A thread pool reuses previously created threads to execute current tasks and offers a solution to the problem of thread cycle overhead and resource thrashing. Since the thread is already existing when the request arrives, the delay introduced by thread creation is eliminated, making the application more responsive.



To use thread pools, we first create a object of ExecutorService and pass a set of tasks to it. ThreadPoolExecutor class allows to set the core and maximum pool size. The runnables that are run by a particular thread are executed sequentially

```
Runnable r1 = new Task("task 1");
```

```
// threads as the fixed pool size(Step 2)
```

```
ExecutorService pool = Executors.newFixedThreadPool(3);
```

```
// passes the Task objects to the pool to execute (Step 3)
```

```
pool.execute(r1);
```

109. Explain Thread Life Cycle?

New: The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

Runnable: The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

Running: The thread is in running state if the thread scheduler has selected it.

Non-Runnable (Blocked): This is the state when the thread is still alive, but is currently not eligible to run.

Terminated: A thread is in terminated or dead state when its run() method exits.

110. What is Thread Synchronization Why we need it?

Thread synchronization is a process of allowing only one thread to access the shared resource. For below scenario we required thread Synchronization.

For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file

111. How Thread Synchronization will work internally?

Synchronization is a process to allow only one thread to be allowed to access the shared resources. If 5 threads trying to access the same resource synchronization will allow the first thread remaining 4 threads will be locked. This locking/monitoring mechanism will be maintained by JVM.

When a thread wants to lock a particular object or class, it asks the JVM. JVM responds to thread with a lock maybe very soon, maybe later, or never. When the thread no longer needs the lock, it returns it to the JVM. If another thread has requested the same lock, the JVM passes the lock to that thread. If a thread has a lock, no other thread can access the locked data until the thread that owns the lock releases it.

JVM maintains a count of the number of times the object has been locked. An unlocked object has a count of zero. When a thread acquires the lock for the first time, the count is incremented to one. Each time the thread acquires a lock on the same object, a count is incremented. Each time the thread releases the lock, the count is decremented. When the count reaches zero, the lock is released and made available to other threads.

112. How to improve the performance in Thread Synchronization?

- ✓ Use Synchronized blocks
- ✓ Keep synchronized methods out of loops if you possibly can.
- ✓ Prioritize threads. Use notify instead of notifyAll. Use synchronization sparingly.

113. What is difference between Executor.submit() and Executor.execute() methods ?

| Execute Method | Submit Method |
|--|--|
| This method is declared in the Executor interface. | This method is declared in the ExecutorService interface. |
| This method can accept only runnable task. | This method can accept both runnable and callable tasks. |
| This method has a return type of void. | This method has a return type of Future. |
| This method is used when we are not bothered about the result but want the code to run in parallel by the worker threads of the thread pool. | This method is used when we care about the result and need it from the task which has been executed. |

114. What is meant by Deadlock?

Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.

115. What is the use of Shutdown Hook?

The shutdown hook can be used to perform cleanup resource or save the state when JVM shuts down normally or abruptly. Performing clean resource means closing log file, sending some alerts or something else. So if you want to execute some code before JVM shuts down, use shutdown hook.

116. How to interrupt the Thread execution?

If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException. If the thread is not in the sleeping or waiting state, calling the interrupt()

method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true.

117. What is the use of volatile keyword in Java?

The Java volatile keyword is used to mark a Java variable as "being stored in main memory". More precisely that means, that every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and that every write to a volatile variable will be written to main memory, and not just to the CPU cache.

It is used to make the class or be thread safe. The volatile keyword can be used either with primitive type or objects.

Volatile keyword is used to modify the value of a variable by different threads.

Example

```
-----  
class Test  
{  
    static int var=5;  
}
```

In the above example, assume that two threads are working on the same class. Both threads run on different processors where each thread has its local copy of var. If any thread modifies its value, the change will not reflect in the original one in the main memory. It leads to data inconsistency because the other thread is not aware of the modified value.

```
class Test  
{  
    static volatile int var =5;  
}
```

In the above example, static variables are class members that are shared among all objects. There is only one copy in the main memory. The value of a volatile variable will never be stored in the cache. All read and write will be done from and to the main memory.

118. Difference b/w volatile & synchronization?

| Volatile | Synchronized |
|---|---|
| volatile keyword can be applied on primitives and objects | Synchronized keyword modifies code blocks and methods. |
| The thread cannot be blocked for waiting in case of volatile. | Threads can be blocked for waiting in case of synchronized. |
| It improves thread performance. | Synchronized methods degrade the thread performance |

It synchronizes the value of one variable at a time between thread memory and main memory.

It synchronizes the value of all variables between thread memory and main memory.

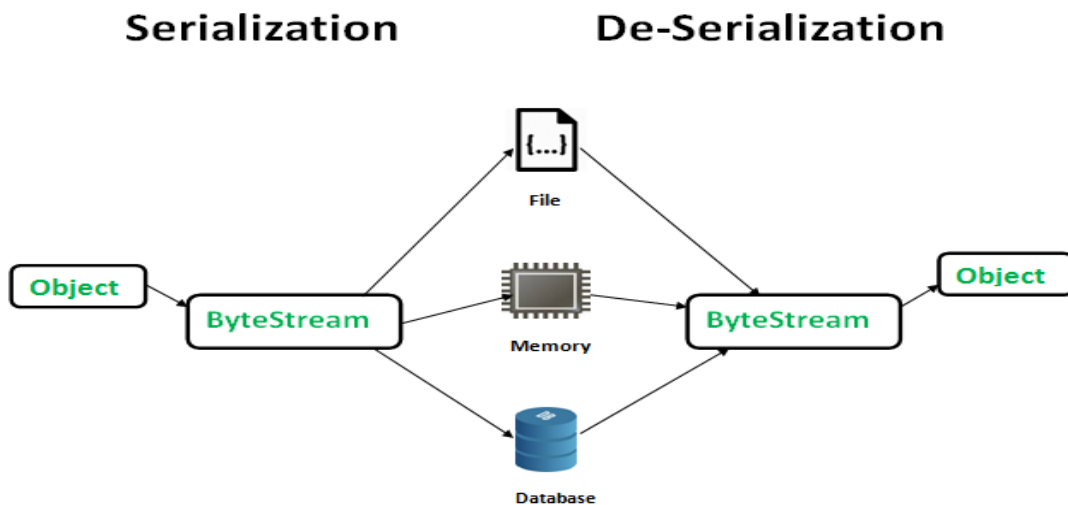
119. What's wrong using HashMap in the multi-threaded environment?

Well, nothing is wrong, depending on how you use it. For example, if you initialize the HashMap just by one thread and then all threads are only reading from it, then it's perfectly fine. One example of this is a Map which contains configuration properties.

The real problem starts when at-least one of that thread is updating HashMap i.e. adding, changing or removing any key value pair. Since put() operation can cause re-sizing and which can further lead to infinite loop, that's why either you should use Hashtable or ConcurrentHashMap, later is better.

120. What is meant by Serialization?

Serialization is a process of writing the object state into stream over the network. Serialization can be achieved by implementing the marker interface Serializable.



Below is the process to make the Object to be serialized in File,


```

class Demo implements java.io.Serializable {
    private static final long serialVersionUID = 6157621820285930845L;
    public int a;
    public String b;

    // Default constructor
    public Demo(int a, String b) {
        this.a = a;
        this.b = b;
    }
}

public class Serialization {
    public static void main(String[] args) throws Exception {
        // Saving of object in a file
        FileOutputStream file = new FileOutputStream("abc.txt");
        ObjectOutputStream out = new ObjectOutputStream(file);

        Demo demo = new Demo(2, "test");

        // Method for serialization of object
        out.writeObject(demo);

        out.close();
        file.close();
    }
}

```

121. What is meant by Deserialization?

The process of getting the existing state of an object is called as Deserialization.

Below is the process to make the Object to be Deserialized in File,

```

class Demo implements java.io.Serializable {
    private static final long serialVersionUID = 6157621820285930845L;
    public int a;
    public String b;

    // Default constructor
    public Demo(int a, String b) {
        this.a = a;
        this.b = b;
    }
}

public class Serialization {
    public static void main(String[] args) throws Exception {
        // Saving of object in a file
        FileInputStream file = new FileInputStream("abc.txt");
        ObjectInputStream in = new ObjectInputStream(file);

        // Method for deserialization of object
        Demo object1 = (Demo)in.readObject();

        System.out.println(object1.getClass());

        in.close();
        file.close();
    }
}

```

122. Why we need the Serialization?

The main purpose of Serialization is to save the state of an object in order to recreate the object when required.

In Java, we create several objects that live and die accordingly, and every object will certainly die when the JVM dies. But sometimes, we might want to reuse an object between several JVMs or we might want to transfer an object to another machine over the network.

Well, serialization allows us to convert the state of an object into a byte stream, which then can be saved into a file on the local disk or sent over the network to any other machine.

The serialization process is platform independent, an object serialized on one platform can be deserialized on a different platform.

123. Why Is Serializable Not Implemented by Object class?

The Object class does not implement Serializable interface because we may not want to serialize all the objects, e.g. serializing a thread does not make any sense because thread running in my JVM would be using my system's memory, persisting it and trying to run it in your JVM would make no sense.

124. What is the use of transient keyword?

- ✓ If we don't want to make any of the variable to be Serializable then will use Transient modifier.
- ✓ Transient is applicable for variables only not the classes and objects.
- ✓ If we use the transient jvm will store the default value instead of actual value.
- ✓ Transient is not applicable for Static variables but transient applicable for instance variables.
- ✓ Transient is not applicable for Final variables because final variable values is assigned at compiled time.

125. Key points to be remembered in Serialization?

- ✓ If a parent class has implemented Serializable interface then child class doesn't need to implement it but vice-versa is not true.
- ✓ Only non-static data members are saved via Serialization process.

- ✓ Static data members and transient data members are not saved via Serialization process. So, if you don't want to save value of a non-static data member then make it transient.
- ✓ Constructor of object is never called when an object is deserialized.
- ✓ Associated objects must be implementing Serializable interface.

126. In which cases will get the NotSerializableException?

If associate class not implementing the Serializable interface will get the NotSerializableException.

For Example

```
Class Student implements Serializable{
    Int id;
    Private Address address;

    //getters and setters
}
```

```
Class Address{
    Int street;

    //getters and setters
}
```

As associated class Address not implementing the Serializable interface during the execution will get NotSerializableException.

127. What is the use of serialization UUID?

The serialization runtime associates with each serializable class a version number, called a serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization.

128. What will happen if we keep the default serialization UUID for all classes?

At the time of deserialization, receiver side JVM will compare the unique ID associated with the Object with local class Unique ID i.e. JVM will also create a Unique ID based on the corresponding .class file which is present in the receiver system. If both unique ID matched then only deserialization will be performed. Otherwise we will get **Runtime Exception saying InvalidClassException.**

129. How to do the custom serialization?

During serialization, there may be data loss if we use the 'transient' keyword. 'Transient' keyword is used on the variables which we don't want to serialize. But sometimes, it is needed to serialize them in a different manner than the default serialization (such as encrypting before serializing etc.), in that case, we have to use custom serialization and deserialization.

In java will use Externalizable interface to achieve the custom serialization. Externalizable interface will have below methods for custom serialization,

```
// to read object from stream  
void readExternal(ObjectInput in)
```

```
// to write object into stream  
void writeExternal(ObjectOutput out)
```

Example:

```
@Override  
public void writeExternal(ObjectOutput out) throws IOException {  
    out.writeObject(name);  
    out.writeInt(age);  
    out.writeInt(year);  
}  
  
@Override  
public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {  
    name = (String) in.readObject();  
    year = in.readInt();  
    age = in.readInt();  
}  
  
Car car = new Car("Shubham", 1995);  
Car newcar = null;  
  
// Serialize the car  
try {  
    FileOutputStream fo = new FileOutputStream("gfg.txt");  
    ObjectOutputStream so = new ObjectOutputStream(fo);  
    so.writeObject(car);  
    so.flush();  
} catch (Exception e) {  
    System.out.println(e);  
}  
  
// Deserializa the car  
try {  
    FileInputStream fi = new FileInputStream("gfg.txt");  
    ObjectInputStream si = new ObjectInputStream(fi);  
    newcar = (Car) si.readObject();  
} catch (Exception e) {  
    System.out.println(e);  
}
```

130. Difference b/w Static, Final, transient and volatile?

| Static | Final | Transient | Volatile |
|---|---|---|--|
| Static variable won't even participate in Serialization. If a variable contains both final and transient keyword here transient modifier will be ignored during deserialization static variable value will loaded from the class | Final variables can be serialized. If a variable contains both final and transient keyword, then it will be serialized. Because here transient modifier will be ignored and final modifier will take over the actions. | Will use transient to make the fields data to be ignored during the Serialization | Volatile will be applicable for multi-threading environment. It is used to store the variable values in System memory instead of Ram memory. System memory will be removed when the System is shut down/forceful restart |

131. If a class is Serializable but its super class is not, what will be the state of the instance variables inherited from super class after deserialization?

Nothing will happen and program will execute perfectly.

```
class Ser_1 {  
    void display(int num) {  
        System.out.println("In Display 1 " + num);  
    }  
}  
  
class Ser_2 extends Ser_1 implements Serializable{  
    private static final long serialVersionUID = 5973259254175630541L;  
    void display(int num) {  
        System.out.println("In Display 1 " + num);  
    }  
}  
  
public class SerilizeException {  
    public static void main(String[] args) {  
        Ser_1 ser1 = new Ser_1();  
        ser1.display(1); //In Display 1 1  
  
        Ser_2 ser2 = new Ser_2();  
        ser2.display(1); //In Display 1 1  
    }  
}
```

- 132. Suppose you have a class which you serialized it and stored in persistence and later modified that class to add a new field. What will happen if you deserialize the object already serialized?**

It depends on whether class has its own serialVersionUID or not. As we know from above question that if we don't provide serialVersionUID in our code java compiler will generate it and normally it's equal to hashCode of object. by adding any new field there is chance that new serialVersionUID generated for that class version is not the same of already serialized object and in this case Java Serialization API will throw java.io.InvalidClassException and this is the reason its recommended to have your own serialVersionUID in code and make sure to keep it same always for a single class.

- 133. Suppose super class of a new class implement Serializable interface, how can you avoid new class to being serialized?**

If Super Class of a Class already implements Serializable interface in Java then its already Serializable in Java, since you cannot unimplemented an interface it's not really possible to make it Non Serializable class but yes there is a way to avoid serialization of new class. To avoid Java serialization you need to implement writeObject() and readObject() method in your Class and need to throw NotSerializableException from those method.

- 134. What is meant by Generics?**

Generics is the new feature introduced in Java 5 to provide the type safe applications.

Without Generics:

```
List list = new ArrayList();  
list.add(10);  
list.add("10");
```

With Generics:

```
With Generics, it is required to specify the type of object we need to store  
List<Integer> list = new ArrayList<Integer>();  
list.add(10);  
list.add("10");// compile-time error
```

With the generics issues will be resolved during the compile time itself.

135. Advantages of Generics?

Type-safety: We can hold only a single type of objects in generics. It doesn't allow to store other objects. Without Generics, we can store any type of objects.

Type casting is not required: There is no need to typecast the object. Before Generics, we need to type cast.

Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime

136. Explain the types of Generics?

Generic Classes:

Like C++, we use <> to specify parameter types in generic class creation. To create objects of generic class, we use following syntax.

```
// We use <> to specify Parameter type
class Test<T>
{
    // An object of type T is declared
    T obj;
    Test(T obj) { this.obj = obj; } // constructor
    public T getObject() { return this.obj; }
}
```

Generic Methods:

We can also write generic functions that can be called with different types of arguments based on the type of arguments passed to generic method, the compiler handles each method.

```
// A Generic method example
static <T> void genericDisplay (T element)
{
    System.out.println(element.getClass().getName() +
        " = " + element);
}
```

T - Type

E - Element

K - Key

N - Number

V - Value

Generics work only with Reference Types:

When we declare an instance of generic type, the type argument passed to the type parameter must be a reference type. We cannot use primitive data types like int,char..,

```
Test<int> obj = new Test<int>(20); //Wrong
```

```
Test<Integer> obj = new Test<Integer>(20); //Correct
```

Wildcard in Java Generics

The ? (question mark) symbol represents the wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number, e.g., Integer, Float, and double. Now we can call the method of Number class through any child class object.

We can use a wildcard as a type of a parameter, field, return type, or local variable. However, it is not allowed to use a wildcard as a type argument for a generic method invocation, a generic class instance creation, or a supertype.

```
abstract class Shape{
    abstract void draw();
}
class Rectangle extends Shape{
    void draw(){System.out.println("drawing rectangle");}
}
class Circle extends Shape{
    void draw(){System.out.println("drawing circle");}
}
class GenericTest{
    //creating a method that accepts only child class of Shape
    public static void drawShapes(List<? extends Shape> lists){
        for(Shape s:lists){
            s.draw();//calling method of Shape class by child class instance
        }
    }
}
```

137. What is Bounded and Unbounded wildcards in Generics ?

Bounded Wildcards are those which impose bound on Type. There are two kinds of Bounded wildcards <? extends T> which impose an upper bound by ensuring that type must be sub class of T and <? super T> where its imposing lower bound by ensuring Type must be super class of T.

This Generic Type must be instantiated with Type within bound otherwise it will result in compilation error. On the other hand <?> represent an unbounded type because <?> can be replaced with any Type.

138. What is the difference between List<? extends T> and List<? super T> ?

Both of List declaration is an example of bounded wildcards, List<? extends T> will accept any List with Type extending T while List<? super T> will accept any List with type super class of T. For example List<? extends Number> can accept List<Integer> or List<Float>.

139. Can we use Generics with Array?

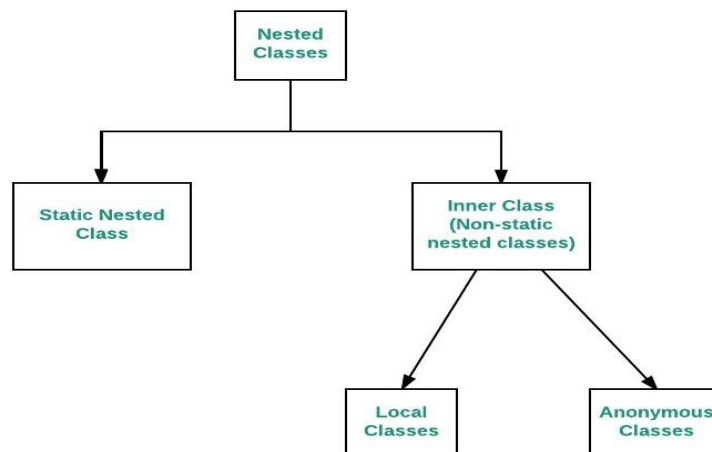
Array doesn't support Generics and that's why Joshua Bloch suggested in Effective Java to prefer List over Array because List can provide compile time type-safety over Array.

140. What is meant by Java Inner classes?

A class which is declared inside another class is called as inner class. Inner classes were introduced in Java 1.1 version.

141. Difference between nested class and inner class in Java?

Inner class is a part of nested class. Non-static nested classes are known as inner classes.



142. What are all the advantages of Inner classes?

- ✓ Modularity

```
class Account{
    class StudentAccount{}
```

```
class PersonalAccount{}  
}
```

We can achieve the modularity by defining the different types of accounts under Account base class

✓ Security

```
class Account{  
private class StudentAccount{  
}
```

In the above example we declared inner class as private class this is how we can achieve security

✓ Code reusability

```
class Account{  
class A{  
class B extends A{  
}
```

In the inner classes we can extend the other inner classes. This is how Reusability can be achieved.

✓ Abstraction

```
class A{  
class B{  
int i =5;  
}  
class C{  
i=i+5;  
}  
}
```

In the above example from B class is not accessible this is the way abstraction will be provided.

143. When we will go for Inner Classes?

Without existing of one class there is no change of existing other class.

```
class car{  
    class engine {}  
}
```

Ex: Without existing car class there is no chance of existing engine class.

Relation b/w outer and inner class is has-a relationship.

144. Explain different types of Inner Classes?

- Member inner class.
- Local inner classes.
- Anonymous inner classes.
- Static inner classes.

145. Explain about member class with an example?

If we are declaring any named class without static modifier inside a class is named as member inner class.

```
class Outer {  
    public void display() {  
        System.out.println("In Outer class display method");  
    }  
  
    class Inner {  
        public void display() {  
            System.out.println("In Inner class display method");  
        }  
    }  
}  
  
public class MemberInnerClass {  
    public static void main(String[] args) {  
        Outer outer = new Outer();  
        outer.display();//In Outer class display method  
  
        //Invoking the Inner class with the help of Outer class.  
        Outer.Inner inner = new Outer().new Inner();  
        inner.display();//In Inner class display method  
    }  
}
```

After compilation will get two class files,

```
Outer.class  
Outer$Inner.class
```

146. Explain Local inner class with an example?

Defining the class inside a method is called as Local Inner class.

```
class Outer1 {
    public void display() {
        class Inner1 {
            public void show() {
                System.out.println("In Inner class display method");
            }
        }

        // Creating the Inner class object outside the Inner class and inside display
        // method
        Inner1 inner1 = new Inner1();
        inner1.show();
    }
}

public class LocalInnerClass {
    public static void main(String[] args) {
        Outer1 outer1 = new Outer1();
        outer1.display(); // In Inner class display method
    }
}
```

147. Explain Anonymous inner class with an example?

A class that does not have any name is called as Anonymous inner classes.

```
interface Age {
    int x = 21;

    void getAge();
}

public class AnonymousDemo {
    public static void main(String[] args) {
        Age age = new Age() {

            @Override
            public void getAge() {
                System.out.print("Age is " + x); //Age is 21
            }
        };
        age.getAge(); //Invoking the Method
    }
}
```

148. Explain about Static inner classes with an example?

Declaring a classes inside a class with static modifier is called as Static Inner Class.

```

class Outer2 {
    public void display() {
        System.out.println("In Outer class display method");
    }

    static class Inner {
        public void display() {
            System.out.println("In Inner class display method");
        }
    }
}

public class StaticInnerClass {

    public static void main(String[] args) {
        Outer2 outer = new Outer2();
        outer.display();//In Outer class display method

        //Invoking the Inner class with the help of Outer class.
        Outer2.Inner inner = new Outer2.Inner();
        inner.display(); ///In Inner class display method
    }
}

```

149. Why we required Anonymous Inner Classes?

If we want to provide the implementation specific to interface/abstract class specific methods then we required Anonymous classes.

150. For interface we can implement methods using Implementation and for Abstract class we can implement methods using concrete class but why we need Anonymous inner class?

If we want to access only the members of interface/abstract class then will go for Anonymous inner class.

151. When we can go for Implementation class for interface and Concrete class for Abstract class?

If we want to provide the implementation for available methods in interface/Abstract class along with Implementation and Concrete class specific methods will go for Implementation class and Concrete class.

152. Is it possible to declare the static variables inside the static inner classes?

Static inner class will allow only static members directly without final keyword

153. Is it possible to declare the static variables inside the non-static inner classes?

In general, Inner classes are not allowing static declarations directly, but static keyword allowed along with final keyword.

154. Can a static nested class have access to the enclosing class non-static methods or instance variables?

No

155. Explain the types of anonymous inner classes?

- ✓ Anonymous inner class that extends a class
- ✓ Anonymous inner class that implements an interface
- ✓ Anonymous inner class as an argument

Anonymous inner class that extends a class:

Here a new keyword is used to create an object of the anonymous inner class that has a reference of parent class type

```
class Car {
    public void engineType() {
        System.out.println("Turbo Engine");
    }
}

public class AnonymousClassDemo {
    public static void main(String args[]) {
        Car c1 = new Car();
        c1.engineType();
        Car c2 = new Car() {
            @Override
            public void engineType() {
                System.out.println("V2 Engine");
            }
        };
        c2.engineType();
    }
}
```

Output

```
Turbo Engine
V2 Engine
```

Anonymous inner class that implements an interface:

Here a new keyword is used to create an object of the anonymous inner class that has a reference of an interface type.

```

interface Software {
    public void develop();
}
public class AnonymousClassDemo1 {
    public static void main(String args[]) {
        Software s = new Software() {
            @Override
            public void develop() {
                System.out.println("Software Developed in Java");
            }
        };
        s.develop();
        System.out.println(s.getClass().getName());
    }
}

```

Output

```

Software Developed in Java
AnonymousClassDemo1$1

```

Anonymous inner class as an argument:

We can use the anonymous inner class as an argument so that it can be passed to methods or constructors.

```

abstract class Engine {
    public abstract void engineType();
}
class Vehicle {
    public void transport(Engine e) {
        e.engineType();
    }
}
public class AnonymousInnerClassDemo2 {
    public static void main(String args[]) {
        Vehicle v = new Vehicle();
        v.transport(new Engine() {
            @Override
            public void engineType() {
                System.out.println("Turbo Engine");
            }
        });
    }
}

```

Output

```

Turbo Engine

```

156. Explain disadvantages of Inner Classes?

Using inner class increases the total number of classes being used by the application. For all the classes created by JVM and loaded in the memory, jvm has to perform some tasks like creating the object of type class. Jvm may have to perform some routine tasks for these extra classes created which may result slower performance if the application is using more number of inner classes.

157. Why a non-final “local” variable cannot be used inside an inner class?

Inner class object cannot use the local variables of the method in which the local inner class is defined.

because use local variables of the method is the local variables of the method are kept on the stack and lost as soon as the method ends.

But even after the method ends, the local inner class object may still be alive on the heap. Method local inner class can still use the local variables that are marked final.

final variable JVM takes these as a constant as they will not change after initiated . And when a inner class try to access them compiler create a copy of that variable into the heap and create a synthetic field inside the inner class so even when the method execution is over it is accessible because the inner class has it own copy.

158. What is an ENUM?

Java Enums are classes that have a fixed set of constants or variables that do not tend to changes. The enumeration in java is achieved by using enum keyword. The java enum constants are static and final implicitly.

159. Why and when to use ENUM?

Use enums when you have values that you know aren't going to change, like month days, days, colors, deck of cards, etc.

160. Difference between Class and ENUM?

An enum can, just like a class, have attributes and methods. The only difference is that enum constants are public, static and final (unchangeable - cannot be overridden).

An enum cannot be used to create objects, and it cannot extend other classes (but it can implement interfaces).

161. Advantages of ENUM?

- ✓ Enums provide the type safety.

- ✓ Enum is easily usable in Switch cases
- ✓ Enum can be traversed.
- ✓ Enum has field's methods and constructors.
- ✓ Enums can implement the interfaces.

162. Can we create the instance for ENUM?

No

163. How do you create Enum without any instance? Is it possible without compile-time error?

Since Enum is viewed as a collection of a well-defined fixed number of instances like Days of Week, Month in a Year, having an Enum without any instance, may seem awkward.

But yes, you can create Enum without any instance in Java, say for creating a utility class. This is another innovative way of using Enum in Java.

```
public enum EnumExample {
    ;// required to avoid compiler error, also signifies no instance

    public static boolean isValid() {
        throw new UnsupportedOperationException("Not supported yet.");
    }
}
```

164. Can we create an instance of Enum outside of Enum itself? If Not, Why?

No, you cannot create enum instances outside of the Enum boundary, because Enum doesn't have any public constructor, and the compiler doesn't allow you to provide any public constructor in Enum. Since the compiler generates a lot of code in response to the enum type declaration, it doesn't allow public constructors inside Enum, which enforces declaring enum instances inside Enum itself.

165. Can we declare Constructor inside Enum in Java?

This is asked along with the previous question on Java Enum. Yes, you can, but remember you can only declare either private or package-private constructor inside enum. public and protected constructors are not permitted inside enum.

166. Difference between ENUM == and equals?

== operator never throws NullPointerException whereas .equals() method can throw NullPointerException.

== is responsible for type compatibility check at compile time whereas .equals() method will never worry about the types of both the arguments.

```

1  enum Day {
2      MON, TUE, WED, THU, FRI, SAT, SUN
3  }
4
5
6  public class EnumExample{
7      public static void main(String[] args) {
8          Day d = null;
9
10         System.out.println(d == Day.MON); //false
11
12         System.out.println(d.equals(Day.MON)); //java.lang.NullPointerException
13     }
14 }

```

167. What does ordinal() method do in Enum?

The ordinal method returns the order in which Enum instances are declared inside Enum. For example in a DayOfWeek Enum, you can declare days in the order they come e.g.

```

public enum DayOfWeek{
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;
}

```

here if we call DayOfWeek.MONDAY.ordinal() it will return 0, which means it's the first instance.

168. Explain the methods inside the Enum?

Values(): The values() method returns an array containing all the values of the enum.

valueOf(): The valueOf() method returns the value of given constant enum.

Ordinal(): The ordinal() method returns the index of the enum value.

169. What is meant I/O streams?

Stream is a process of writing and reading the data. I/O streams provide the classes to store and read the data from files

170. Explain about Standard/Default streams?

Standard Input – This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as System.in.

Standard Output – This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as System.out.

Standard Error – This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as System.err

171. Explain the different types of Streams?

Depending upon the data a stream holds, it can be classified into:

Byte Stream: Byte stream is used to read and write a single byte (8 bits) of data.

Character Stream: Character stream is used to read and write a single character of data.

172. Explain different types of Byte Streams?

Input Streams:

=====

FileInputStream : The FileInputStream class of the java.io package can be used to read data (in bytes) from files.

ByteArrayInputStream: The ByteArrayInputStream class of the java.io package can be used to read an array of input data (in bytes).

ObjectInputStream: The ObjectInputStream class of the java.io package can be used to read objects that were previously written by ObjectOutputStream.

FilterInputStream

i.BufferedInputStream: The BufferedInputStream class of the java.io package is used with other input streams to read the data (in bytes) more efficiently.

ii.DataInputStream : Java DataInputStream class allows an application to read primitive data from the input stream

Output Streams:

=====

FileOutputStream : The FileOutputStream class of the java.io package can be used to write data (in bytes) to the files.

ByteArrayOutputStream : The ByteArrayOutputStream class of the java.io package can be used to write an array of output data (in bytes).

ObjectOutputStream: The ObjectOutputStream class of the java.io package can be used to write objects that can be read by ObjectInputStream.

FilterOutputStream

i.BufferedOutputStream: The BufferedOutputStream class of the java.io package is used with other output streams to write the data (in bytes) more efficiently.

ii.DataOutputStream: Java DataOutputStream class allows an application to write primitive data from the input stream

173. Explain about Character Streams?

1. Reader

BufferedReader: The BufferedReader class of the java.io package can be used with other readers to read data (in characters) more efficiently.

InputStreamReader: The InputStreamReader class of the java.io package can be used to convert data in bytes into data in characters.

FileReader: The FileReader class of the java.io package can be used to read data (in characters) from files.

StringReader: The StringReader class of the java.io package can be used to read data (in characters) from strings.

2. Writer

BufferedWriter: The BufferedWriter class of the java.io package can be used with other writers to write data (in characters) more efficiently.

OutputStreamWriter: The OutputStreamWriter class of the java.io package can be used to convert data in character form into data in bytes form.

FileWriter: The FileWriter class of the java.io package can be used to write data (in characters) to files.

StringWriter: The StringWriter class of the java.io package can be used to write data (in characters) to the string buffer.

174. What is meant by Reflection?

Java Reflection is a process of examining or modifying the run time behavior of a class at run time.

175. Give a real time example for Reflection?

Javac
Javap
java

Above commands will use the Reflections to compile, run and compile the Java programs. Here above will read the variables, methods by using the Reflections API and executing those during the execution time.

176. Advantages of Reflections?

- ✓ We can change the runtime behaviour of any class
- ✓ We can access the private fields of an class

177. Disadvantages of Reflection?

- ✓ Performance
- ✓ Security
- ✓ Unexpected crashes as we are changing the data during the runtime.

178. How do you access the package of a class?

The package of a class can be accessed by calling the method `getPackage()` on the class object.

```
Class myClass = Class.forName('java.lang.String');  
Package package = myClass.getPackage();
```

179. How do you access the interfaces implemented by a class?

The interfaces of a class can be accessed by calling the method `getInterfaces()` on the class object.

```
Class myClass = Class.forName('java.lang.String');  
Package package = myClass.getInterfaces();
```

180. How do you access the parent class of a class?

The parent or super class of a class can be accessed by calling the method `getSuperClass()` on the class object.

```
Class myClass = Class.forName('java.lang.String');  
Package package = myClass.getSuperclass();
```

181. How do you retrieve class access modifiers reflection?

Class access modifiers are the access modifiers such as public, private etc. that a class is declared with. Class modifiers can be accessed calling the method `getModifiers()` on the class object.

```
Class myClass = Class.forName('java.lang.String');  
int modifier = myClass.getModifiers();
```

182. How do you access constructors defined in a class using reflection?

Constructors of a class can be accessed by calling the method `getConstructors()` on the class object.

```
Class myClass = Class.forName('java.lang.String');  
int modifier = myClass.getConstructors ();
```

183. How do you access fields defined in a class using reflection?

Fields of a class can be accessed by calling the method `getFields()` on the class object.

```
Class myClass = Class.forName('java.lang.String');  
Field[] fields = myClass.getFields();
```

184. How do you access annotations defined in a class using reflection?

Annotations of a class can be accessed by calling the method `getAnnotations()` on the class object.

```
Class myClass = Class.forName('java.lang.String');  
Annotation[] annotations = myClass.getAnnotations();
```

185. How to access the private methods, Fields in a class?

Accessing the Private fields,

```
Class myClass = Class.forName('java.lang.String');  
Field privateField = myClass.getDeclaredField("fieldInfo");  
privateField.setAccessible(true);
```

Accessing the Private Methods,

```
Class myClass = Class.forName('java.lang.String');  
Method privateMethod = myClass.getDeclaredMethod("fieldInfo");  
privateMethod.setAccessible(true);
```

186. Difference between instanceof and isInstance()?

Instanceof: It's an operator. instanceof operator must be evaluated to a type (class or interface), which is known at compile time.

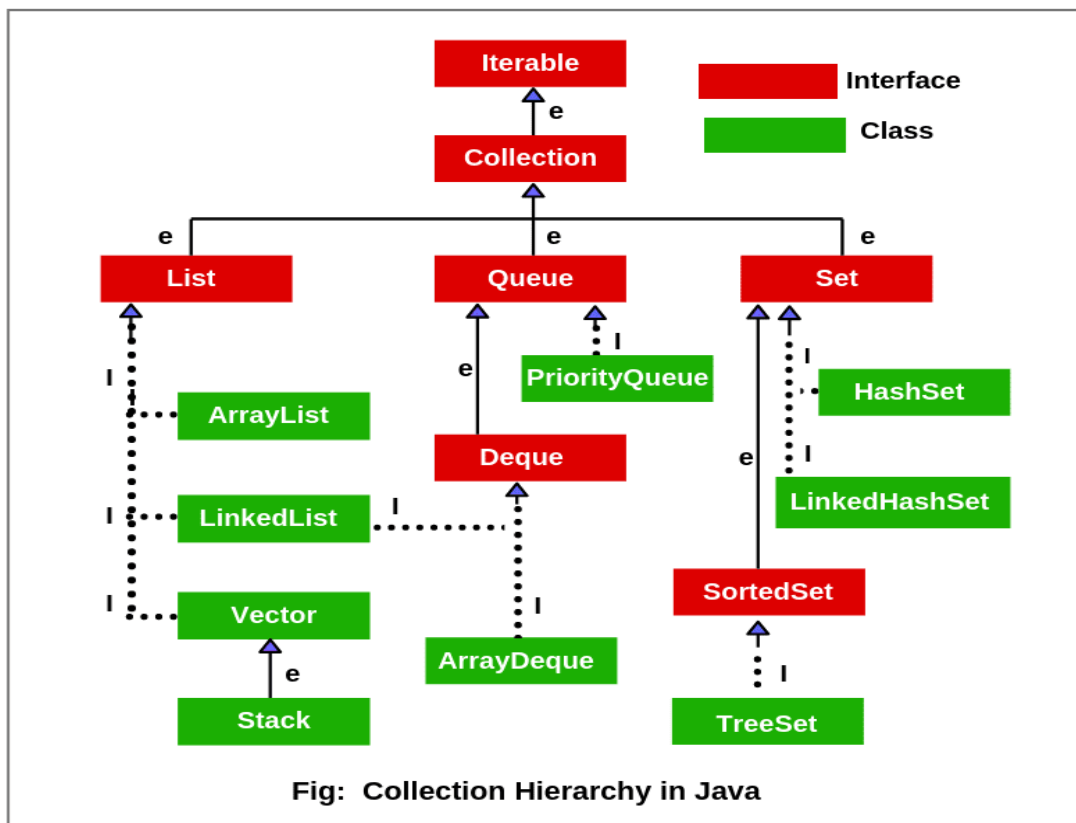
instanceof operator throws compile time error (Incompatible conditional operand types) if we check object with other classes which it doesn't instantiate.

```
Integer i = 5;  
System.out.println(i instanceof String); //Incompatible conditional operand types Integer and String
```

isInstance(): It's a method. It will evaluate the type of (class or interface) during the Run time. If the object is matching it will return true otherwise false.

187. Explain Collections hierarchy?

Set of classes and interfaces which can be used to represent the group of objects as single entity is known as Collections.



Iterable: This is the root interface for the entire collection framework. The collection interface extends the Iterable interface. Therefore, inherently, all the interfaces and classes implement this interface. The main functionality of this interface is to provide an iterator for the collections. Therefore, this interface contains only one abstract method which is the iterator.

Collection: This interface extends the Iterable interface and is implemented by all the classes in the collection framework. This interface contains all the basic methods which every collection has like adding the data into the collection, removing the data, clearing the data

List: Which we can store all the ordered collection of the objects. This also allows duplicate data to be present in it.

This list interface is implemented by various classes.

ArrayList: It's a dynamic growable array. Which is used to store the ordered collections of objects. This will allow duplicates and it is not synchronized.

Methods: add, addAll, remove, set, clear, size, contains, get, isEmpty

```
List<Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
list.remove(1);
list.get(0);
list.size();
list.set(0, 2);
list.contains(1);
```

Vector: It's like an ArrayList but only the difference is it's synchronized because of it's slower in performance.

Methods: add, addAll, remove, set, size, clear, contains, get, isEmpty

```
List<Integer> list = new Vector<>();
list.add(1);
list.add(2);
list.remove(1);
list.get(0);
list.size();
list.set(0, 2);
list.contains(1);
```

Stack: Stack class models and implements the Stack data structure. The class is based on the basic principle of last-in-first-out. In addition to the basic push and pop operations, the class provides three more functions of empty, search and peek

```
Stack<String> stack = new Stack<>();
stack.push("Geeks");
stack.push("For");
stack.pop();
```

LinkedList: LinkedList class is an implementation of the LinkedList data structure which is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses.

```
LinkedList<Integer> ll = new LinkedList<>();
ll.add(1);
ll.remove(1);
```

Set: A set is an unordered collection of objects in which duplicate values cannot be stored. This collection is used when we wish to avoid the duplication of the objects and wish to store only the unique objects.

HashSet: The objects that we insert into the HashSet do not guarantee to be inserted in the same order. The objects are inserted based on their hashCode. This class also allows the insertion of NULL elements

Methods: add, clone, remove, clear, contains, size, isEmpty

```
Set<String> hashSet = new HashSet<>();  
hashSet.add(null);  
hashSet.add("test");  
hashSet.add("java");  
hashSet.add("test");  
hashSet.add("code");  
  
System.out.println(hashSet);//[null, java, code, test]
```

LinkedHashSet: It's same as HashSet but it will maintain the insertion order. It will accept null values to.

Methods: add, clone, remove, clear, contains, size, isEmpty

```
Set<String> hashSet = new LinkedHashSet<>();  
hashSet.add(null);  
hashSet.add("test");  
hashSet.add("java");  
hashSet.add("test");  
hashSet.add("code");  
  
System.out.println(hashSet);//[null, test, java, code]
```

SortedSet: This interface is very similar to the set interface. The only difference is that this interface has extra methods that maintain the ordering of the elements.

TreeSet: It will maintain the ascending order while storing and it does not allow the null values.

Methods: add, addAll, clone, remove, clear, contains, size, isEmpty

```
Set<String> hashSet = new TreeSet<>();  
//hashSet.add(null); It will not accept the Null values  
hashSet.add("test");  
hashSet.add("java");  
hashSet.add("test");  
hashSet.add("code");  
  
System.out.println(hashSet);//[code, java, test]
```

Queue: As name suggest a queue interface maintains the FIFO (First In First Out).

For example, whenever we try to book a ticket, the tickets are sold at the first come first serve basis. Therefore, the person whose request arrives first into the queue gets the ticket.

Priority Queue: A Priority Queue is used when the objects are supposed to be processed based on the priority. It is known that a queue follows the First-In-First-Out algorithm, but sometimes the elements of the queue are needed to be processed according to the priority and this class is used in these cases. The elements of the priority queue are ordered according to the natural ordering.

```
Queue<Integer> pQueue = new PriorityQueue<>();
pQueue.add(10);
pQueue.add(20);
pQueue.add(15);

System.out.println(pQueue);//[10, 20, 15]

System.out.println(pQueue.peek()); //10
System.out.println(pQueue.poll()); //10
System.out.println(pQueue.peek()); //15
```

Deque: This is a very slight variation of the queue data structure. Deque, also known as a double-ended queue, is a data structure where we can add and remove the elements from both the ends of the queue.

ArrayDeque: ArrayDeque is dynamic growable array. This is a special kind of array that grows and allows users to add or remove an element from both sides of the queue. Array dequeues have no capacity restrictions and they grow as necessary to support usage.

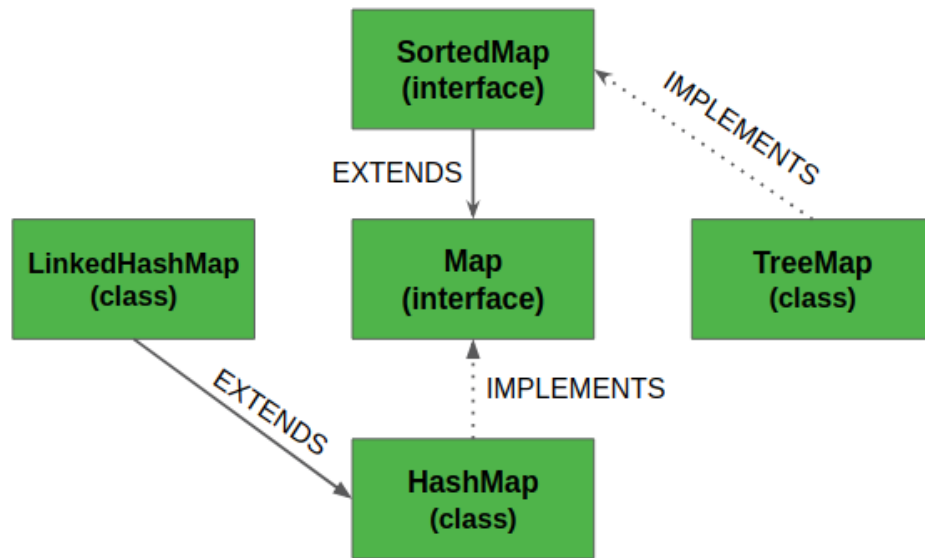
```
ArrayDeque<Integer> deque = new ArrayDeque<>();
deque.add(10);
deque.add(20);
deque.add(15);

System.out.println(deque);// [10, 20, 15]

deque.clear();

deque.addFirst(564);
deque.addLast(24);
System.out.println(deque);// [564, 24]
```

Map: It's a two dimensional array. Which is used to store the key and value pairs. **The Map interface is not a subtype of the Collection interface.**



MAP Hierarchy in Java

HashMap: HashMap is the implementation of Map, but it doesn't maintain any order. It will accept only one Null key but any number of Null Values. It does not allow the duplicate keys but allow duplicate values. It doesn't not maintain the order.

Methods: clear, containsKey, containsValue, entrySet, get, keyset, put, remove, size, values

```
Map<String, Integer> hm = new HashMap<>();

hm.put(null, 100);
hm.put("b", 200);
hm.put("c", 300);
hm.put("d", 400);
hm.put(null, 500);

System.out.println(hm); //{null=100, b=200, c=300, d=400}
```

LinkedHashMap: It's like a HashMap but it will maintain the Insertion order.

Methods: clear, containsKey, containsValue, entrySet, get, keyset, put, remove, size, values

```

Map<String, Integer> hm = new LinkedHashMap<>();

hm.put(null, null);
hm.put("d", 400);
hm.put("b", 200);
hm.put("c", 300);
hm.put(null, 500);

System.out.println(hm); //{null=500, d=400, b=200, c=300}

```

SortedMap: It's like a map but it does not allow the Null Keys and it will sort the data based on the Key.

TreeMap: TreeMap is the implementation of Map and SortedMap. It maintains ascending order. but it does not allow the Null Keys and it will sort the data based on the Key.

Methods: clear, containsKey, containsValue, entrySet, get, keyset, put, remove, size, values

```

Map<String, Integer> hm = new TreeMap<>();

//hm.put(null, null); //java.lang.NullPointerException:
hm.put("d", 400);
hm.put("b", 200);
hm.put("c", 300);

System.out.println(hm); //{b=200, c=300, d=400}

```

HashTable: It will also store the key and value pairs. It does not allow duplicate key, null key and null values. It is synchronized. It does not maintain the insertion order.

```

Hashtable<String, Integer> hm = new Hashtable<>();

hm.put("d", 400);
hm.put("b", 200);
//hm.put("c", null); //java.lang.NullPointerException
//hm.put(null, 300); //java.lang.NullPointerException

System.out.println(hm); //{b=200, d=400}

```

Properties: The properties object contains key and value pair both as a string. The java.util.Properties class is the subclass of Hashtable. It can be used to get property value based on the property key. The Properties class provides methods to get data from the properties file and store data into the properties file.

```

FileReader reader=new FileReader("db.properties");

Properties p=new Properties();
p.load(reader);

System.out.println(p.getProperty("user"));
System.out.println(p.getProperty("password"));

```

188. Difference between List, Set and Map?

| List | Set | Map |
|---|---|---|
| It's an index based | It's not index based | Its key based |
| It will preserve the order of insertion | It will not preserve the order of insertion | It will not preserve the order of insertion |
| It will allow duplicates | It won't allow duplicates | It won't allow duplicate keys |
| It's a part of collections | It's not a part of collections | It's not a part of collections |

189. Why Map is not a part of Collections?

- ✓ Each Collection stores a single value where as a Map stores key-value pair. So methods in Collection interface are incompatible for Map interface. For example in Collection we have add(Object o). What would be such implementation in Map. It doesn't make sense to have such a method in Map. Instead we have a put(key,value) method in Map.
- ✓ Also if you recall Collection interface implemented Iterable interface i.e any interface with .iterator() method should return an iterator which must allow us to iterate over the values stored in the Collection. Now what would such method return for a Map? Key iterator or a Value iterator? This does not make sense either.

190. Difference between ArrayList and Vector?

| ArrayList | Vector |
|---|--|
| It's not synchronized | Its Synchronized |
| It's not thread safe | It's thread safe |
| It is more efficient | It is less efficient |
| It is suitable for single thread applications | It is suitable for multi thread applications |

191. Difference between ArrayList and LinkedList?

| ArrayList | Linked List |
|--|--|
| Its best in searching | Its best in adding and deletion |
| It will use dynamic array internally | It will use doubly linked list internally |
| It is not synchronized | It is not synchronized |
| An ArrayList class can act as a list only because it implements List only. | LinkedList class can act as a list and queue both because it implements List and Deque interfaces. |

192. Difference between HashSet and TreeSet?

| HashSet | TreeSet |
|---|--|
| HashSet maintains no order | TreeSet maintains ascending order |
| It will accept the Null values | It does not allow null values |
| Hash set is implemented using HashTable | The tree set is implemented using a tree structure. |
| Hash set use equals method to compare two objects | Tree set use compare method for comparing two objects. |

193. Difference between HashMap and TreeMap?

| HashMap | TreeMap |
|---|---|
| HashSet maintains no order | TreeSet maintains ascending order |
| It will accept one Null Key | It does not allow null key |
| Hash set is implemented using HashTable | The tree set is implemented using a tree structure. |

194. Difference between HashMap and HashTable?

| HashMap | HashTable |
|---|--|
| It's not synchronized | It's synchronized |
| It will be used in single thread application | It will be useful in multi-threaded applications |
| Performance is high | Performance is low |
| It will accept one null key and multiple null values. | It will not accept any null key and values |

195. Difference between Collection and Collections?

| Collection | Collections |
|---|--|
| It's an interface | It's a class. |
| Its provides interfaces and classes for to implement over the List, Set | It's provide the utility methods for sorting, synchronizing. |

196. Difference between Array and Collection?

| Array | Collection |
|--|---|
| Arrays are always of fixed size | Collection is dynamic growable Array |
| User can not increase or decrease the length of the array according to their requirement or at runtime | Here the size will grow automatically. |
| Arrays can only store homogeneous or similar type objects | Collection, heterogeneous objects can be stored |

197. Difference between Iterator and ListIterator?

| Iterator | ListIterator |
|--|--|
| The Iterator traverses the elements in the forward direction only. | ListIterator traverses the elements in backward and forward directions both. |
| The Iterator can be used in List, Set, and Queue | ListIterator can be used in List only. |
| If we try to remove the data while traversing will get ConcurrentModificationException | Can easily add elements to a collection at any time. |

198. Difference between Iterator and Enumeration?

| Iterator | Enumeration |
|--|---|
| It can be used with any class of the collection framework. | It can be used only with legacy class of the collection framework such as a Vector and HashTable. |
| Any changes in the collection, such as removing element from the collection during a thread is iterating collection then it throw concurrent modification exception. | Enumeration is Fail safe in nature. It doesn't throw concurrent modification exception |
| The Iterator can perform read and remove operation while traversing the collection. | The Enumeration can perform only read operation on the collection. |

```
List list = new ArrayList(Arrays.asList(new String[] { "Apple", "Cat", "Dog", "Rat" }));
Vector v = new Vector(list);

Enumeration e = v.elements();
while (e.hasMoreElements()) {
    System.out.println(e.nextElement());
}

Iterator i = v.iterator();
while (i.hasNext()) {
    System.out.println(i.next());
}
```

199. Difference between Comparator and Comparable?

| Comparable | Comparator |
|---|---|
| Its provide the Natural Ordering | Its provide the Custom ordering |
| Its presents under java.lang package | Its present under java.util package |
| It provides one method named compareTo(). | It provides one method named compare(). |

200. Why Collection interface does not extend Cloneable and Serializable interface?

Collection is an interface that specifies a group of objects known as elements. The details of how the group of elements is maintained is left up to the concrete implementations of Collection. For example, some Collection implementations like List allow duplicate elements whereas other implementations like Set don't.

Many Collection implementations (including all of the ones provided by the JDK) will have a public clone method. For example, what does it mean to clone a Collection that's backed by a terabyte SQL database? Should the method call cause the company to requisition a new disk farm? Similar arguments hold for serializable.

201. What is the use of IdentityHashMap and WeakHashMap?

IdentityHashMap: is similar to HashMap except that it uses reference equality when comparing elements. It is mandates the use of the equals() method when comparing objects.

WeakHashMap: is an implementation of the Map interface that stores only weak references to its keys. Storing only weak references allows a key-value pair to be garbage collected when its key is no longer referenced outside of the WeakHashMap.

202. How to design a good key for hashmap?

It's always good key object must provide same hashCode() again and again, no matter how many times it is fetched. Similarly, same keys must return true when compare with equals() method and different keys must return false.

For this reason, immutable classes are considered best candidate for HashMap keys.

203. What are different Collection views provided by Map interface?

Map interface provides 3 views of key-values pairs stored in it:

- ✓ key set view
- ✓ value set view
- ✓ entry set view

All the views can be navigated using iterators.

204. How to make a collection read only?

```
List<String> list = new ArrayList<>();
list.add("test");

Set<String> set = new HashSet<>();
set.add("test");

Map<String, Integer> map = new HashMap<>();
map.put("test", 100);

Collections.unmodifiableList(list);
Collections.unmodifiableSet(set);
Collections.unmodifiableMap(map);
```

205. How to make collections thread safe?

```
List<String> list = new ArrayList<>();
list.add("test");

Set<String> set = new HashSet<>();
set.add("test");

Map<String, Integer> map = new HashMap<>();
map.put("test", 100);

Collections.synchronizedList(list);
Collections.synchronizedSet(set);
Collections.synchronizedMap(map);
```

206. What are different ways to iterate over a list?

```
List<String> list = new ArrayList<>();
list.add("test");

//Iterator
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}

// for loop
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}

// for loop advance
for (String temp : list) {
    System.out.println(temp);
}

// while loop
int j = 0;
while (j < list.size()) {
    System.out.println(list.get(j));
    j++;
}
```

207. What is difference between fail-fast and fail-safe?

When trying to remove the element from list during the iteration will get the `ConcurrentModificationException`, This case is called as fail-fast.

To avoid the above issue `CopyOnWriteArrayList` is introduced. it will not perform the operations on original list it will create a copy and perform the operations on it. this is called fail-safe

208. What is the advantage of Properties file?

If you change the value in the properties file, you don't need to recompile the java class. So, it makes the application easy to manage. It is used to store information which is to be changed frequently.

209. What is hash-collision in Hashtable and how it is handled in Java?

Two different keys with the same hash value are known as hash-collision. Two separate entries will be kept in a single hash bucket to avoid the collision. There are two ways to avoid hash-collision.

- ✓ Separate Chaining
- ✓ Open Addressing

210. Define dictionary class?

The Dictionary class is a Java class that has a capability to store key-value pairs.

211. Explain UnsupportedOperationException?

`UnsupportedOperationException` is an exception which is thrown on methods that are not supported by actual collection type.

For example, Developer is making a read-only list using "`Collections.unmodifiableList(list)`" and calling `call()`, `add()` or `remove()` method. It should clearly throw `UnsupportedOperationException`.

212. Name the collection classes that gives random element access to its elements?

Collection classes that give random element access to its elements are: 1) `ArrayList`, 2) `HashMap`, 3) `TreeMap`, and 4) `Hashtable`.

213. Explain the methods of iterator interface?

public boolean hasNext(): It returns true in the iterator has elements; otherwise, it returns false.

public Object next(): This method returns the element and moves the pointer to the next value.

public void remove(): This Java method can remove the last elements returned by the iterator. Public void remove() is less used.

214. Explain Map.entry In Map?

This method returns a view of the collection. For example, consider cityMap as a map. The developer can use entrySet() to get the set view of map having an element Map.Entry. Programmer can also use getKey() and getValue() of the Map.Entry to get the pair of key and value of the map.

215. Explain the best practices in Java Collection Framework?

- ✓ Choose the correct type of collection depends on the need.
- ✓ Avoid rehashing or resizing by estimating the total number of elements to be stored in collection classes.
- ✓ Write a Java program in terms of interfaces. This will help the developer to change it's implementation effortlessly in the future.
- ✓ A developer can use Generics for type-safety.
- ✓ Use immutable classes given by the Java Development Kit. Avoid implementation of equals() and hashCode() for custom classes.
- ✓ A programmer should use the Collections utility class for algorithms or to get read-only, synchronized, or empty collections. This will enhance code reusability with low maintainability.

216. How to convert ArrayList to Array and Array to ArrayList?

```
String [] strArr = new String [] { "a", "c", "d"};

//Arrays to List
List<String> list = Arrays.asList(strArr);
System.out.println(list); //[a, c, d]

ArrayList<String> al=new ArrayList<>();
al.add("ankit");
al.add("nippun");

//ArrayList to Array
String [] arr = al.toArray(new String [al.size()]);

for(String str: arr) {
    System.out.println(str);//ankit nippun
}
```

217. Write a java program to display the data in list as descending order?

```

List<Integer> list = new ArrayList<>();
list.add(10);
list.add(50);
list.add(30);

Collections.sort(list);
Collections.reverse(list);

System.out.println(list);//[50, 30, 10]

```

218. Write an example for Java Comparable Interface?

```

class Employee implements Comparable<Employee> {
    int rollno;
    String name;
    @Override
    public int compareTo(Employee arg0) {
        return Integer.compare(rollno, arg0.getRollno());
    }
    public int getRollno() {
        return rollno;
    }
    public void setRollno(int rollno) {
        this.rollno = rollno;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}

public class CollectionsExample {
    public static void main(String[] args) {
        List<Employee> list = new ArrayList<>();
        Employee employee = new Employee();
        employee.setName("Test");
        employee.setRollno(103);
        Employee employee1 = new Employee();
        employee1.setName("Sunny");
        employee1.setRollno(102);
        list.add(employee);
        list.add(employee1);
        Collections.sort(list);

        for(Employee emp: list) {
            System.out.println(emp.getRollno() + " " + emp.getName()); //102 Sunny103 Test
        }
    }
}

```

219. Write an example to remove the duplicates from ArrayList?

```

public static void main(String[] args) {
    List<Integer> intList = new ArrayList<>();
    intList.add(10);
    intList.add(20);
    intList.add(20);
    intList.add(30);
    intList.add(10);

    Set<Integer> intSet = new HashSet<>(intList);
    System.out.println(intSet); // [20, 10, 30]
}

```

220. Explain the ways to iterate the Map?

```

Map<Integer, String> map = new HashMap<>();
map.put(101, "Sunny");
map.put(102, "Bunny");

// Iterator
Iterator<Map.Entry<Integer, String>> itr = map.entrySet().iterator();
while (itr.hasNext()) {
    Map.Entry<Integer, String> entry = itr.next();
    System.out.println("Key = " + entry.getKey() + ", Value = " + entry.getValue());
}

// For Loop
for (Map.Entry<Integer, String> entry : map.entrySet()) {
    System.out.println("Key = " + entry.getKey() + ", Value = " + entry.getValue());
}

// forEach(action) method to iterate map
map.forEach((k, v) -> System.out.println("Key = " + k + ", Value = " + v));

```

221. How ArrayList will work internally?

When you initialize an ArrayList, an array of size 10 (default capacity) is created and an element added to the ArrayList is actually added to this array. 10 is the default size and it can be passed as a parameter while initializing the ArrayList.

When adding a new element, if the array is full, then a new array of 50% more the initial size is created and the last array is copied to this new array so that now there are empty spaces for the new element to be added.

Since the underlying data-structure used is an array, it is fairly easy to add a new element to the ArrayList as it is added to the end of the list. When an element is to be added anywhere else, say the beginning, then all the elements shall have to move one position to the right to create an empty space at the beginning for the new element to be added. This process is time-consuming (linear-time). But the Advantage of ArrayList is that retrieving an element at any position is very fast (constant-time), as underlying it is simply using an array of objects.

Formula for calculating whether list size exceeding the 50%,

```
int newCapacity = (oldCapacity * 3)/2 + 1;
```

222. How HashMap works internally?

HashMap internally works on principle of Hashing.

Hashing in its simplest form, is a way to assigning a unique code for any variable/object after applying any formula/algorithm on its properties. "Hash function should return the same hash code each and every time when the function is applied on same or equal objects.

HashMap internally uses the static nested class Entry to store the key value pair. Entry class stores the key value pairs internally in LinkedList.

When an Entry object needs to be stored in particular index, HashMap checks whether there is already an entry?? If there is no entry already present, the entry object is stored in this location.

If there is already an object sitting on calculated index, its next attribute is checked. If it is null, and current entry object becomes next node in linkedlist. If next variable is not null, procedure is followed until next is evaluated as null.

What if we add the another value object with same key as entered before. Logically, it should replace the old value. How it is done? Well, after determining the index position of Entry object, while iterating over linkedlist on calculated index, HashMap calls equals method on key object for each entry object.

The initial capacity of hashmap is=16

The default load factor of hashmap=0.75

According to the formula as mentioned above: $16 * 0.75 = 12$

It represents that 12th key-value pair of hashmap will keep its size to 16. As soon as 13th element (key-value pair) will come into the Hashmap, it will increase its size from default $24 = 16$ buckets to $25 = 32$ buckets.